Fast 3D Graphics in Processing for Android

By Andres Colubri

October 3rd
UIC Innovation Center
Chicago

7. OpenGL and Processing

OpenGL ES

3D drawing in Android is handled by the GPU (Graphic Processing Unit) of the device.

The most direct way to program 3D graphics on Android is by means of OpenGL ES.

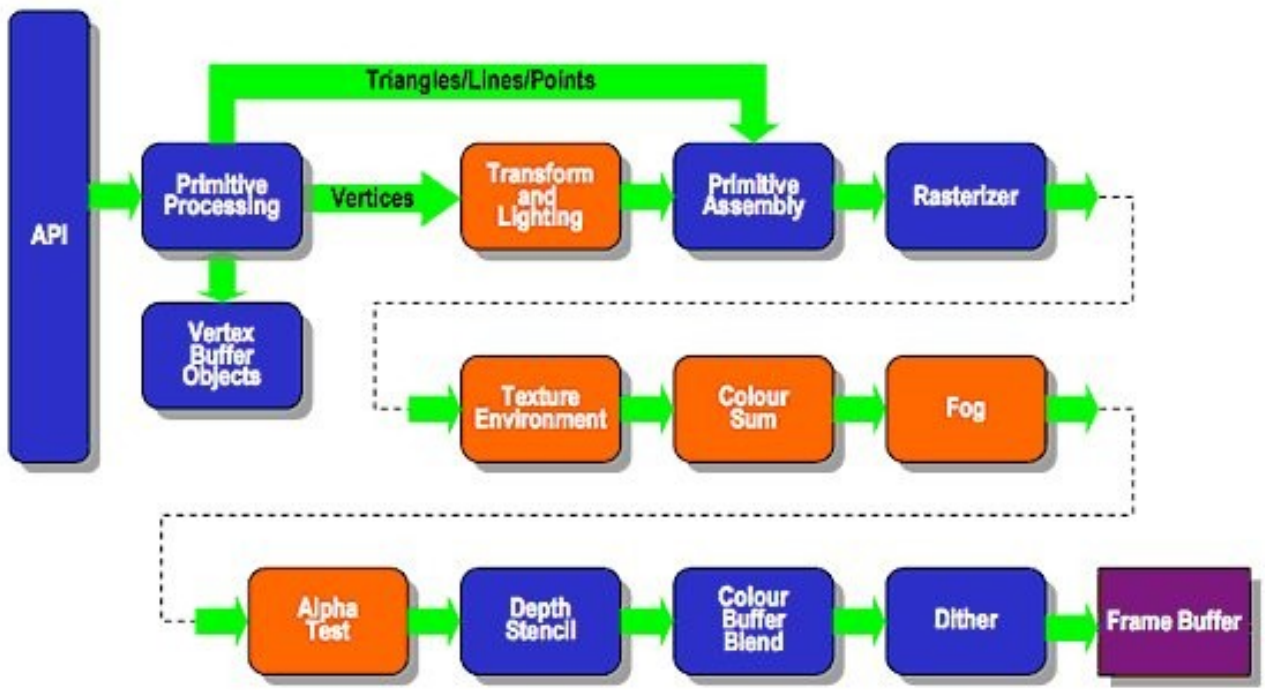
OpenGL ES is a cross-platform API for programming 2D and 3D graphics on embedded devices (consoles, phones, appliances, etc).

OpenGL ES consists in a subset of OpenGL.

http://developer.android.com/guide/topics/graphics/opengl.html
http://www.khronos.org/opengles/

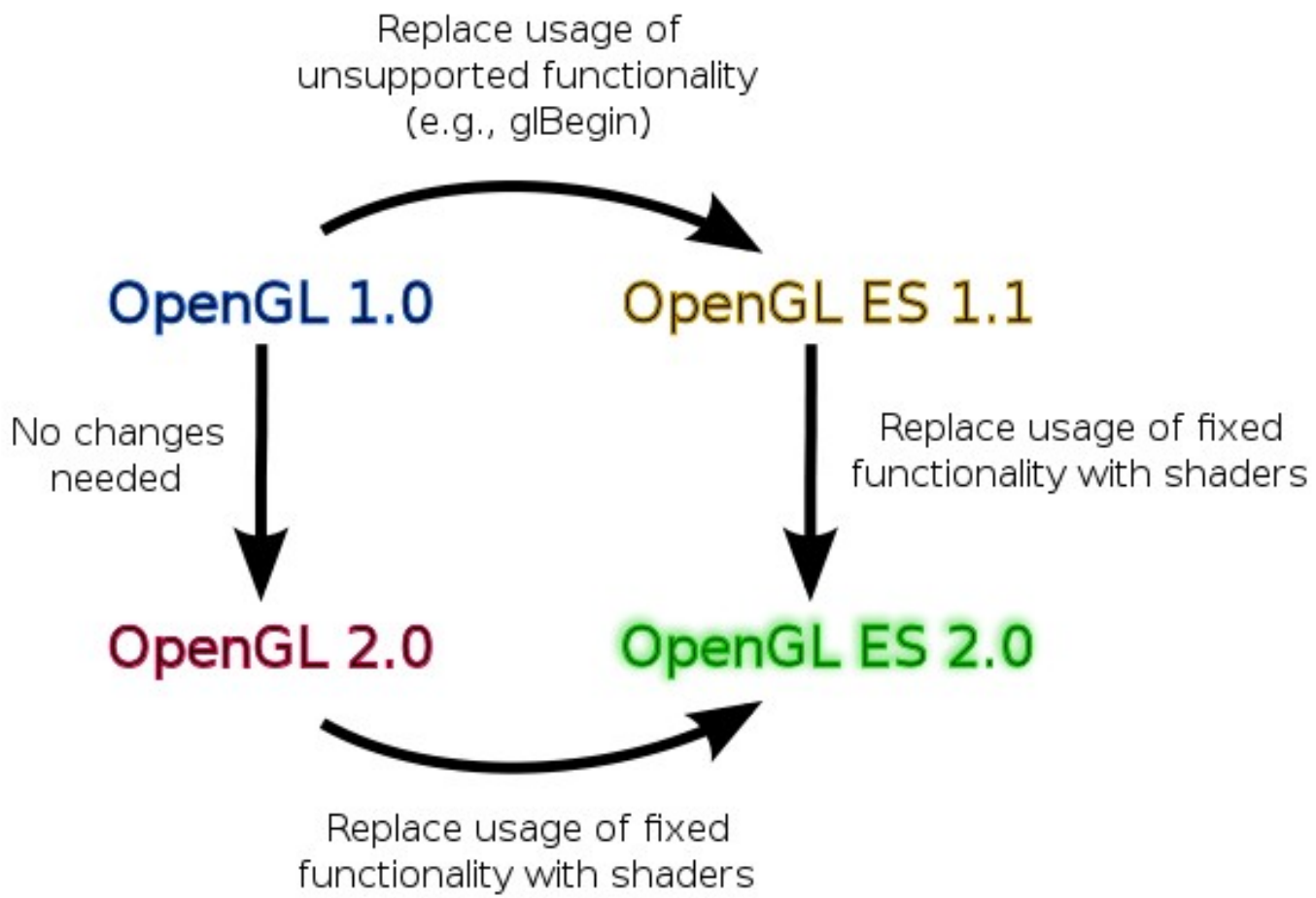
OpenGL ES is the 3D API for other platforms, such as Nokia and iPhone:

The graphics pipeline on the GPU



The graphics pipeline is the sequence of steps in the GPU from the data (coordinates, textures, etc) provided through the OpenGL ES API to the final image on the screen (or Frame Buffer)

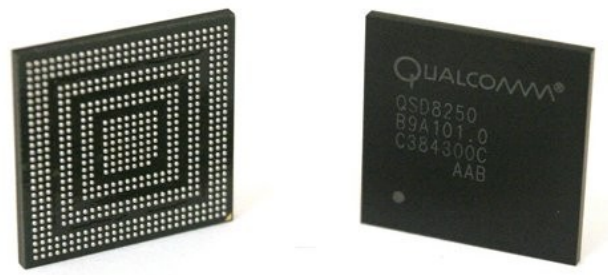
Relationship between OpenGL and OpenGL ES



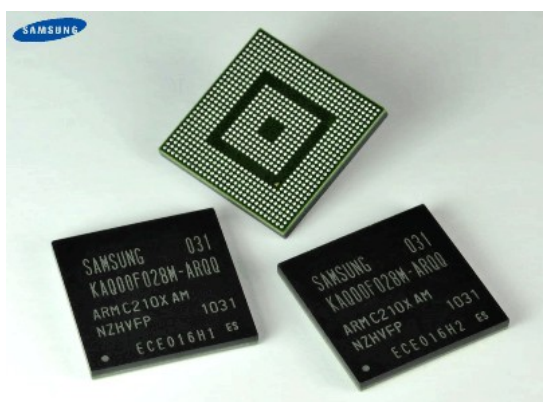
Source: http://wiki.maemo.org/OpenGL-ES

Mobile GPUs

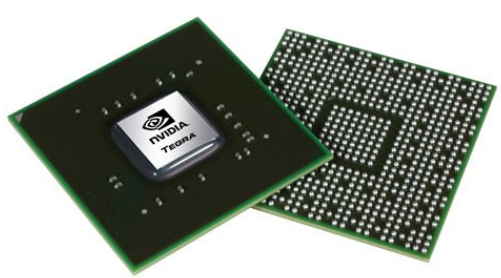
The main GPUs manufacturers for mobile devices are Qualcomm, PowerVR and NVidia.



Qualcomm Adreno



PowerVR SGX



NVidia Tegra

Comparison of current mobile GPUs

	Medium performance	High performance
Qualcomm Adreno	200 (Nexus 1, HTC Evo, Desire)	205 (Desire HD)
PowerVR SGX	SGX 530 (Droid, Droid 2, DroidX)	SGX 540 (Galaxy S, Vibrant, Captivate)
Nvidia Tegra		Tegra 250

Useful websites for benchmark information about mobile GPUs:
http://smartphonebenchmarks.com/
http://www.glbenchmark.com/latest_results.jsp?

Hardware requirements for 3D in Processing **for Android**

1. In principle, any GPU that supports OpenGL ES 1.1

2. GPUs such as the Adreno 200 or PowerVR SVG 540/530 are recommended.

3. Older GPUs found on devices such as the G1 might work, but the performance is limited.

4. As a general requirement for Processing, Android 2.1. However, certain OpenGL features were missing in 2.1, so froyo (2.2) is needed to take full advantage of the hardware.

The A3D renderer

1. In Processing for Android there is no need to use OpenGL ES directly (although it is possible).

2. The drawing API in Processing uses OpenGL internally when selecting the A3D (Android 3D) renderer.

3. The renderer in Processing is the module that executes all the drawing commands.

4. During the first part of this workshop we used the A2D renderer, which only supports 2D drawing.

5. The renderer can be specified when setting the resolution of the output screen with the size() command:
   size(width, height, renderer)
   where renderer = A2D or A3D

6. If no renderer is specified, then A2D is used by default.

Offscreen drawing

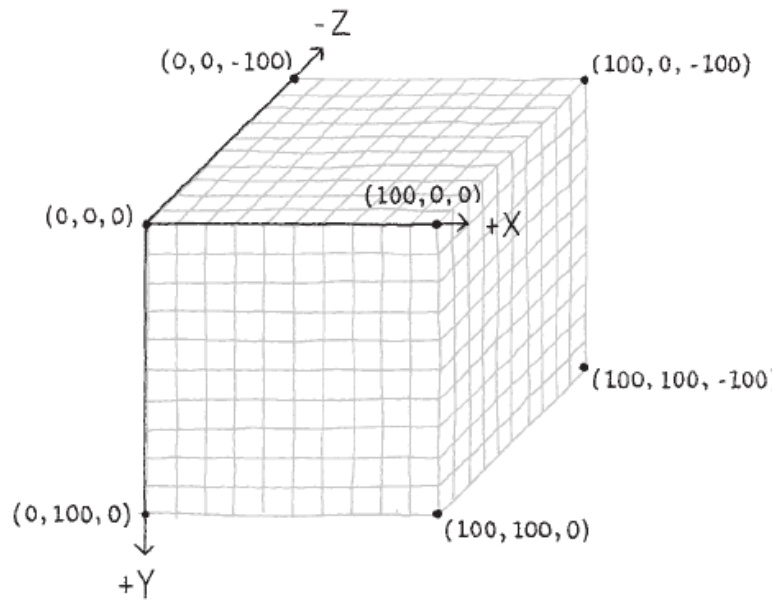
We can create an offscreen A3D surface by using the createGraphics() method:

```
PGraphicsAndroid3D pg;

void setup() {
    size(480, 800, A3D);
    pg = createGraphics(300, 300, A3D);
    ...
}
```

The offscreen drawing can be later used as an image to texture an object or to combine with other layers. We will see more of this at the end.

```
void draw() {
  pg.beginDraw();
  pg.rect(100, 100, 50, 40);
  pg.endDraw();

  ...
  cube.setTexture(pg.getOffscreenImage());
  ...
}
```

8. Geometrical transformations



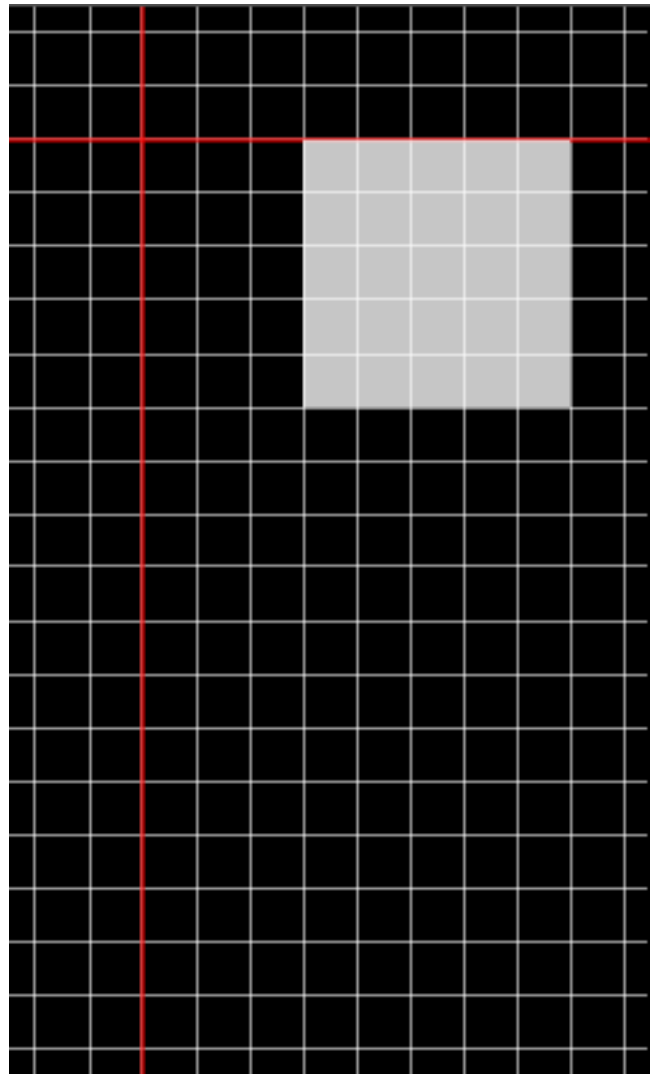
Casey Reas and Ben Fry.
<Getting Started with Processing>.
O'Really Media, 2010

1. The coordinate system in Processing is defined with the X axis running from left to right, Y axis from top to bottom and negative Z pointing away from the screen.
2. In particular, the origin is at the upper left corner of the screen.
3. Geometrical transformations (translations, rotations and scalings) are applied to the entire coordinate system.

Translations

The translate(dx, dy, dz) function displaces the coordinate system by the specified amount on each axis.

```
void setup() {
  size(240, 400, A3D);
  stroke(255, 150);
}

void draw() {
  background(0);

  translate(50, 50, 0);

  noStroke();
  fill(255, 200);
  rect(60, 0, 100, 100);
}
```

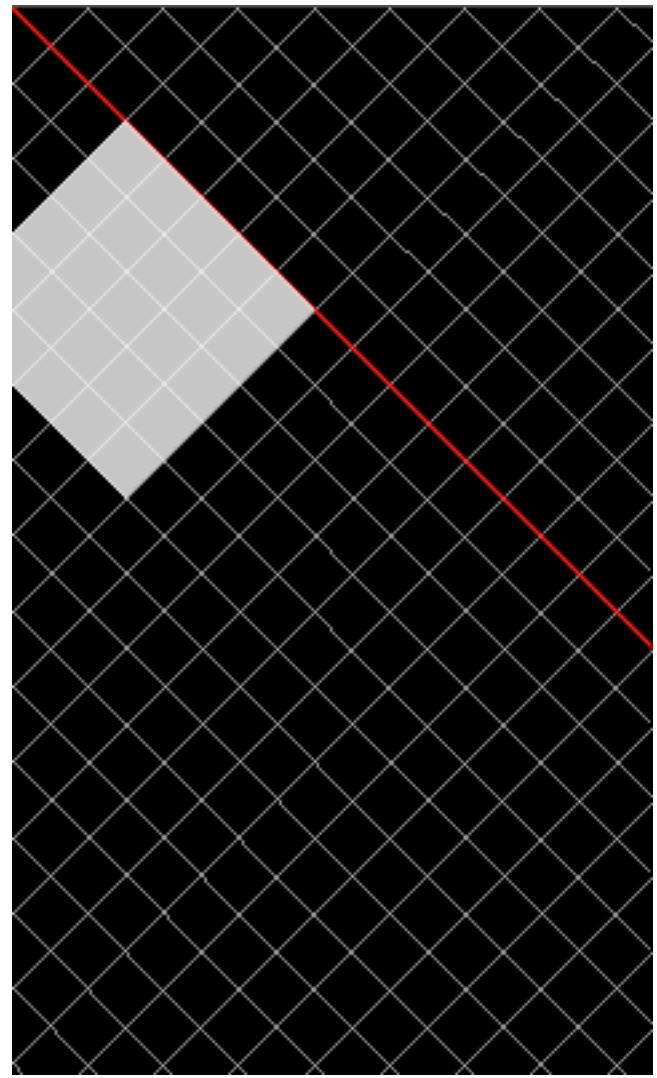


Rotations

Rotations have always a rotation axis that passes through the origin of the coordinate system. This axis could be the X, Y, Z axis, or an arbitrary vector:

```
rotateX(angle), rotateY(angle),
rotateZ(angle), rotate(angle, vx,
vy, vz)
```

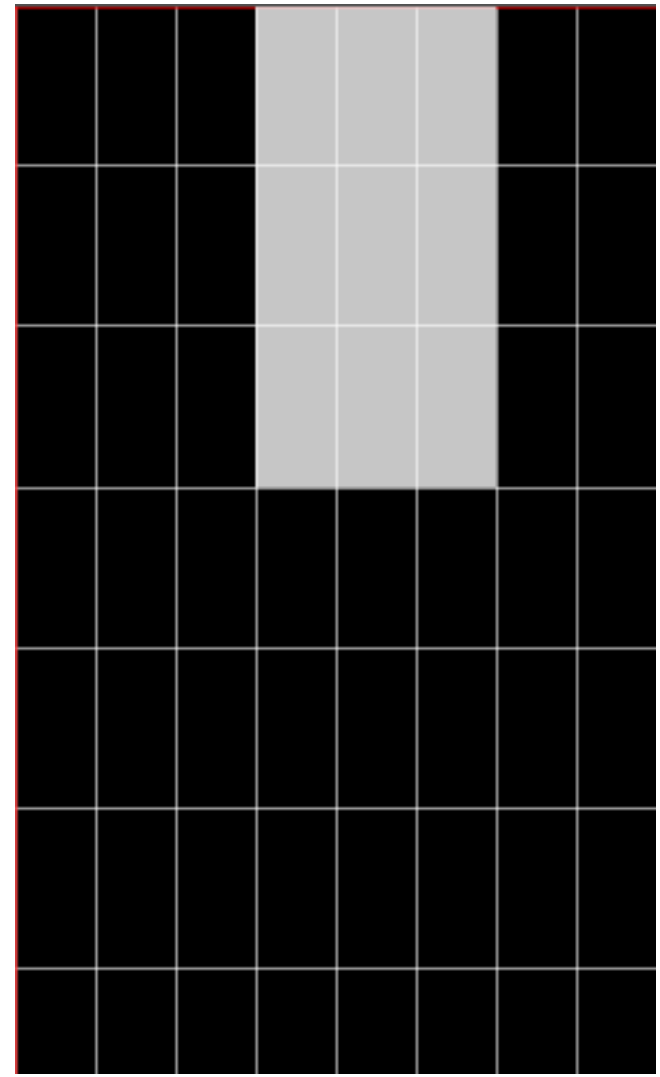
```
void setup() {
  size(240, 400, A3D);
  stroke(255, 150);
}

void draw() {
  background(0);
  rotateZ(PI / 4);
  noStroke();
  fill(255, 200);
  rect(60, 0, 100, 100);
}
```



Scaling

Scaling can be uniform (same scale factor on each axis) or not, since the scale(sx, sy, sz) function allows to specify different factors along each direction.

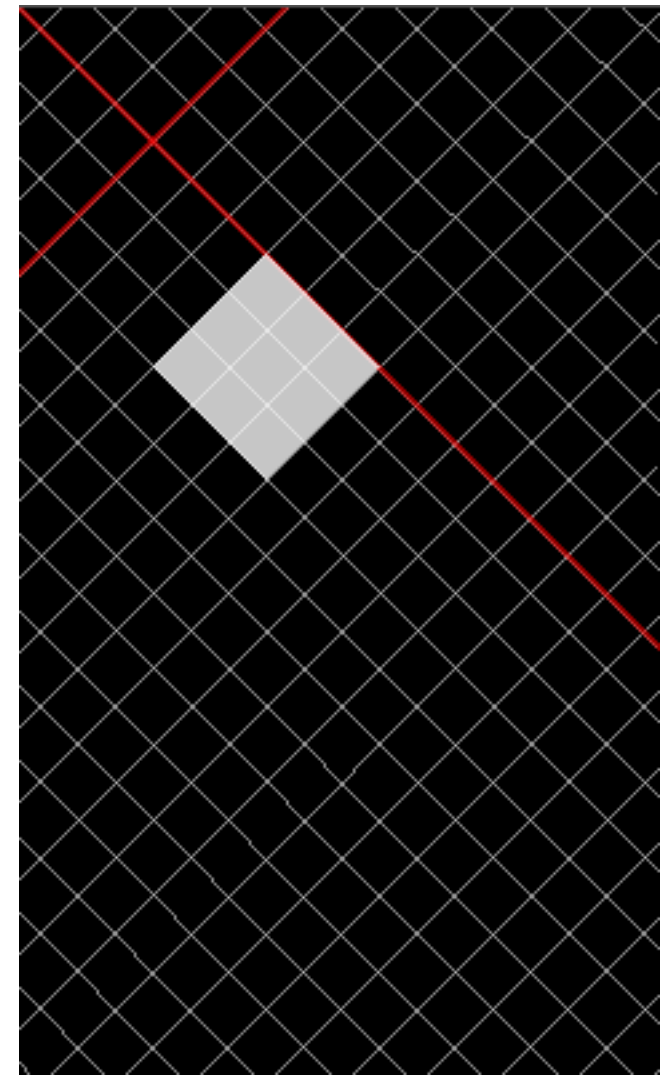
```
void setup() {
  size(240, 400, A3D);
  stroke(255, 150);
}

void draw() {
  background(0);
  scale(1.5, 3.0, 1.0);
  noStroke();
  fill(255, 200);
  rect(60, 0, 60, 60);
}
```



Just a couple of important points about geometrical transformations...

1. By combining translate() with rotate(), the rotations can be applied around any desired point.
2. The order of the transformations is important

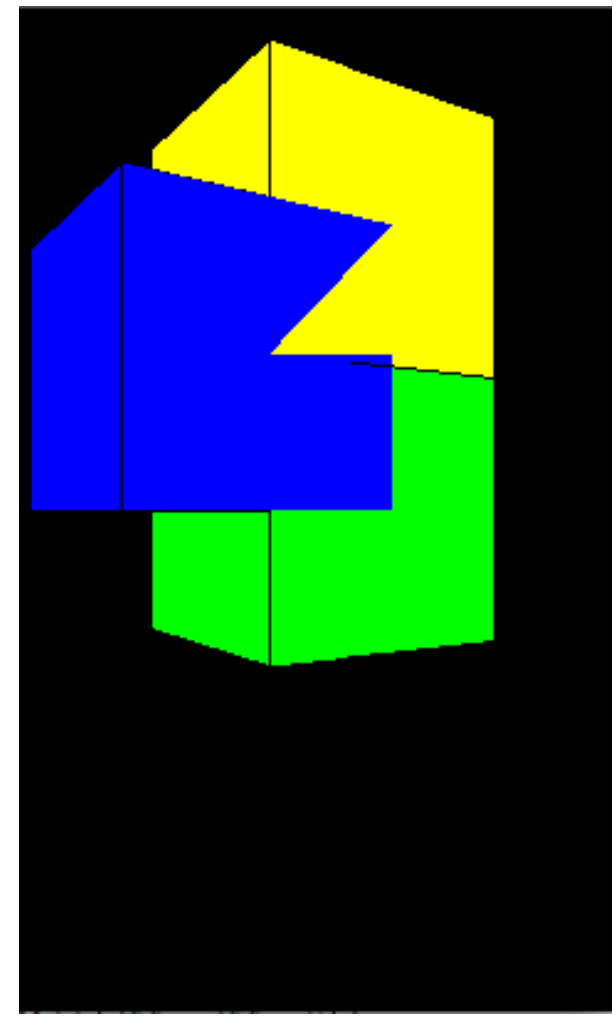
```
void setup() {
  size(240, 400, A3D);
  stroke(255, 150);
}

void draw() {
  background(0);
  translate(50, 50, 0);
  rotateZ(PI / 4);
  noStroke();
  fill(255, 200);
  rect(60, 0, 60, 60);
}
```



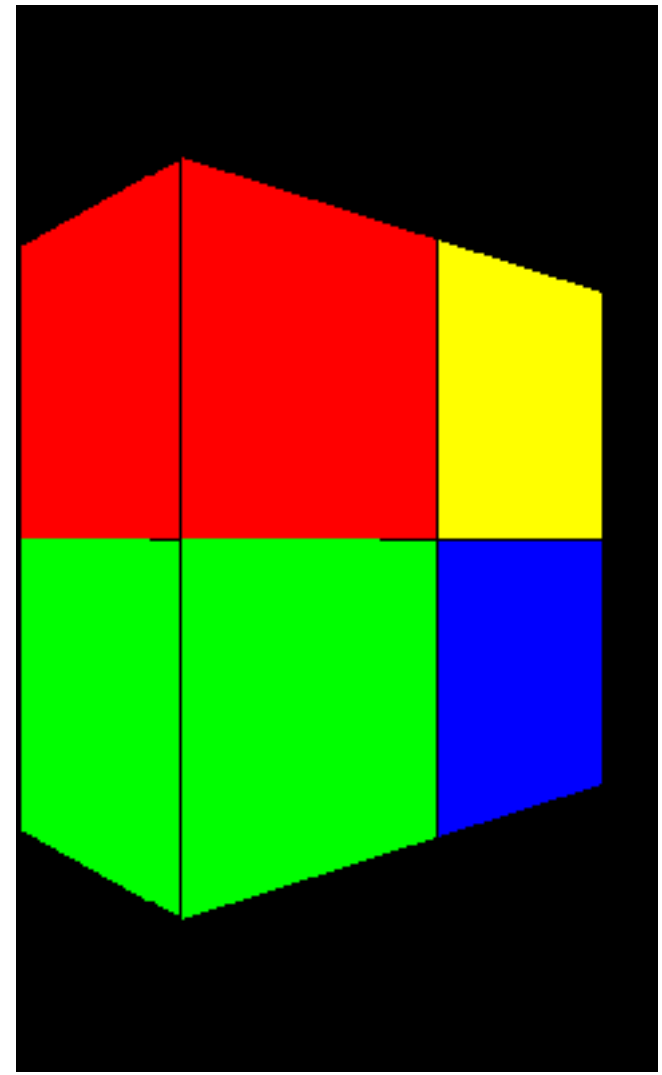
The transformation stack

1. The transformation stack we have in the 2D mode is also available in A3D through the functions pushMatrix() and popMatrix().
2. All the geometric transformations issued between two consecutive calls to pushMatrix() and popMatrix() will not affect the objects drawn outside.

```
void setup(){
  size(240, 400, A3D);
}
void draw(){
  background(0);
  translate(width/2, height/2);
  rotateY(frameCount*PI/60);
  translate(-50, -50);
  fill(255, 0, 0);
  box(100, 100, 100);
  translate(50, -50);
  fill(255, 255, 0);
  box(100, 100, 100);
  translate(-50, 50);
  fill(0, 0, 255);
  box(100, 100, 100);
  translate(50, 50);
  fill(0, 255, 0);
  box(100, 100, 100);
}
```

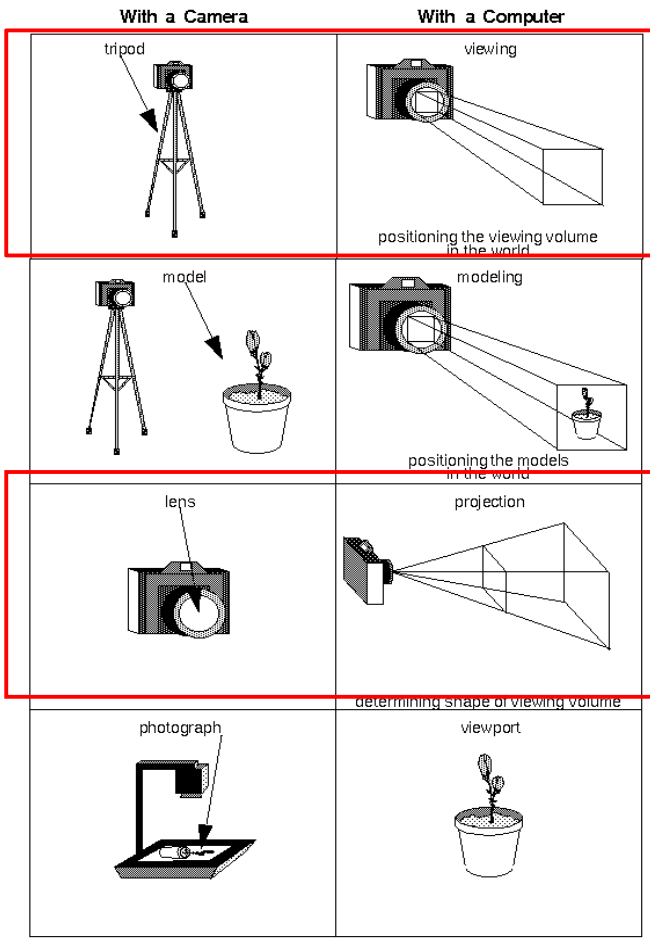


```
void setup(){
  size(240, 400, A3D);
}
void draw(){
  background(0);
  translate(width/2, height/2);
  rotateY(frameCount*PI/60);
  pushMatrix();
  translate(-50, -50);
  fill(255, 0, 0);
  box(100, 100, 100);
  popMatrix();
  pushMatrix();
  translate(50, -50);
  fill(255, 255, 0);
  box(100, 100, 100);
  popMatrix();
  pushMatrix();
  translate(50, 50);
  fill(0, 0, 255);
  box(100, 100, 100);
  popMatrix();
  pushMatrix();
  translate(-50, 50);
  fill(0, 255, 0);
  box(100, 100, 100);
  popMatrix();
}
```



9. Camera and perspective

1. Configuring the view of the scene in A3D requires setting the camera location and the viewing volume.
2. This can be compared with setting a physical camera in order to take a picture:




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

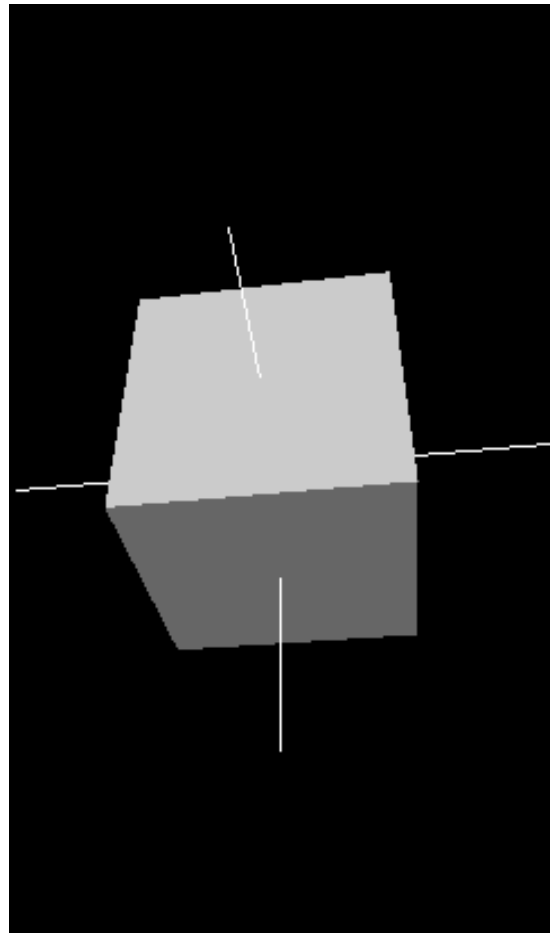
```
perspective(fov, aspect, zNear, zFar)
ortho(left, right, bottom, top, near, far)
```

(image from the OpenGL Red Book, first edition)

Camera placement

1. The camera placement is specified by the eye position, the center of the scene and which axis is facing upwards:camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
2. If camera() is not called, A3D automatically does it with the following values: width/2.0, height/2.0, (height/2.0) / tan(PI*60.0 / 360.0), width/2.0, height/2.0, 0, 0, 1, 0

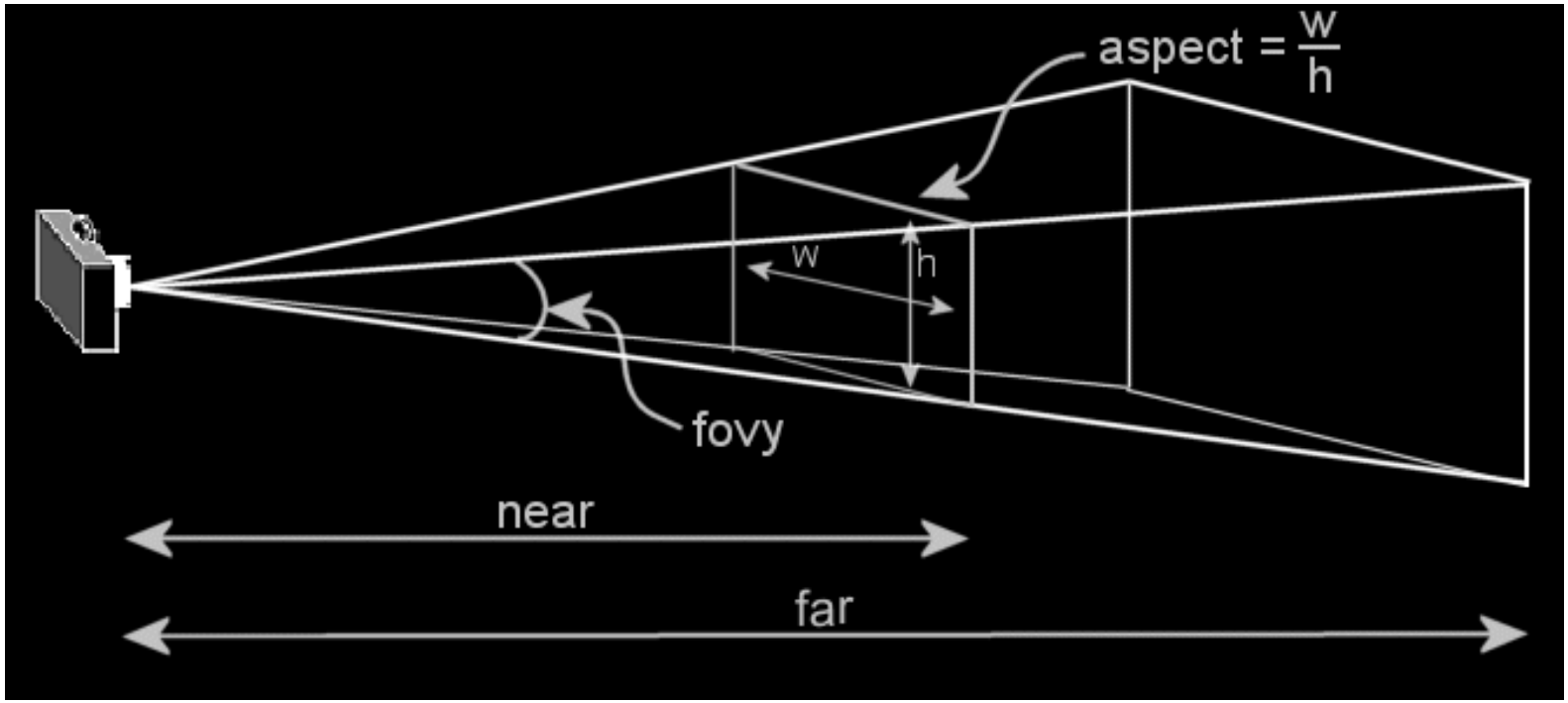
```
void setup() {
  size(240, 400, A3D);
  fill(204);
}

void draw() {
  lights();
  background(0);
  camera(30.0, mouseY, 220.0,
         0.0, 0.0, 0.0,
         0.0, 1.0, 0.0);
  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);
}
```



Perspective view

The viewing volume is a truncated pyramid, and the convergence of the lines towards the eye point create a perspective projection where objects located farther away from the eye appear smaller.

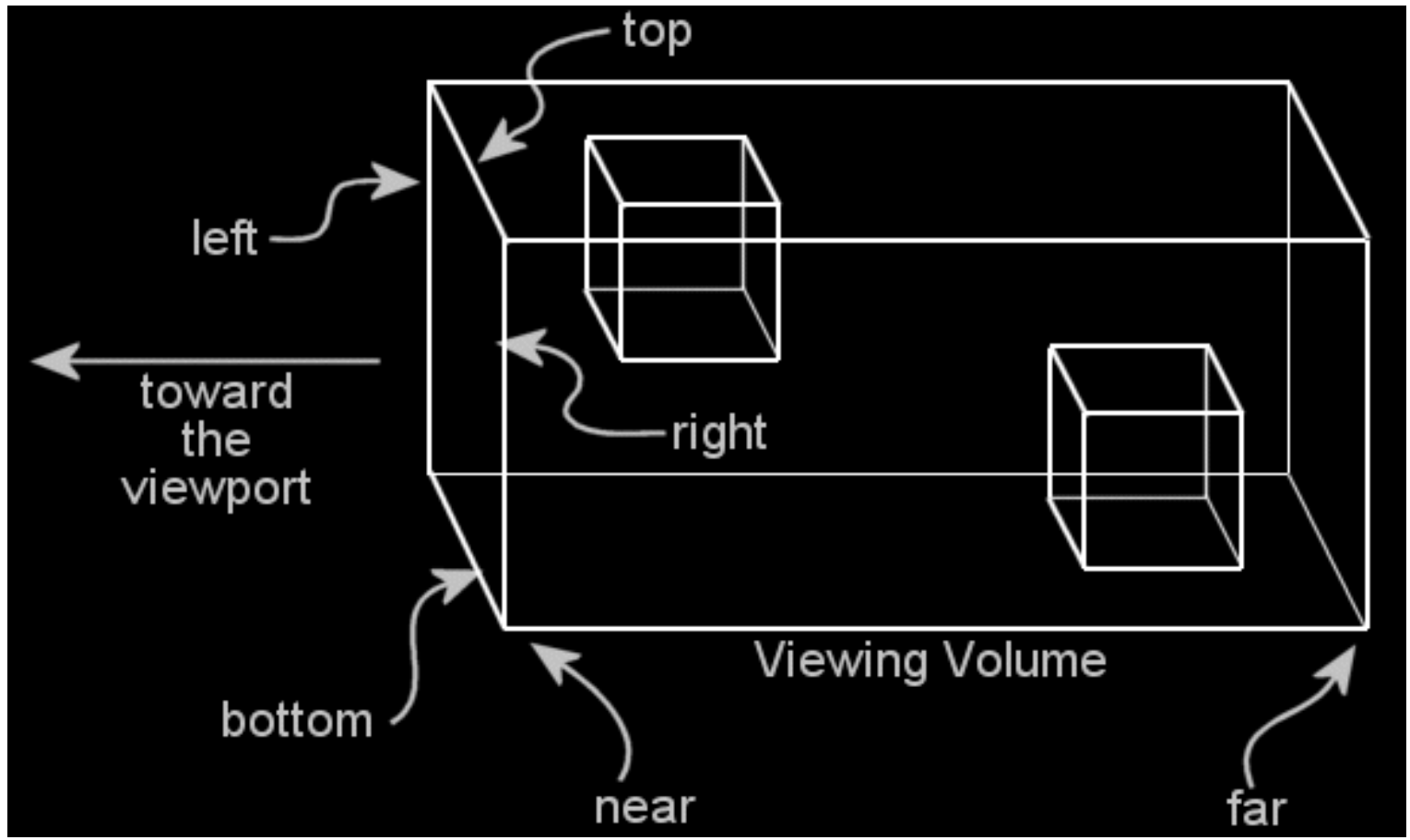
from http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson1.php



perspective(fov, aspect, zNear, zFar)
perspective(PI/3.0, width/height, cameraZ/10.0, cameraZ*10.0) where cameraZ is ((height/2.0) / tan(PI*60.0/360.0)) (default values)

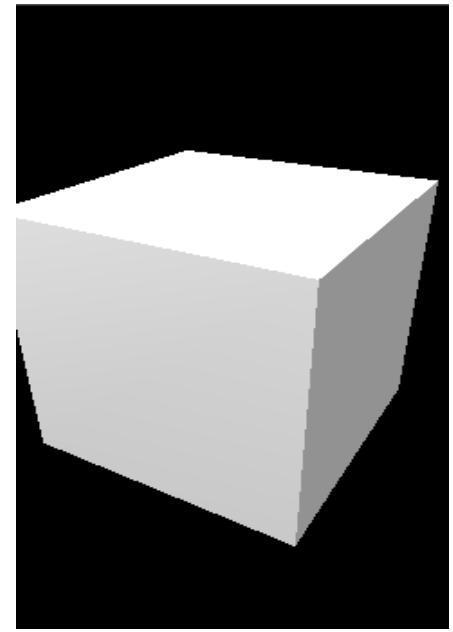
Orthographic view

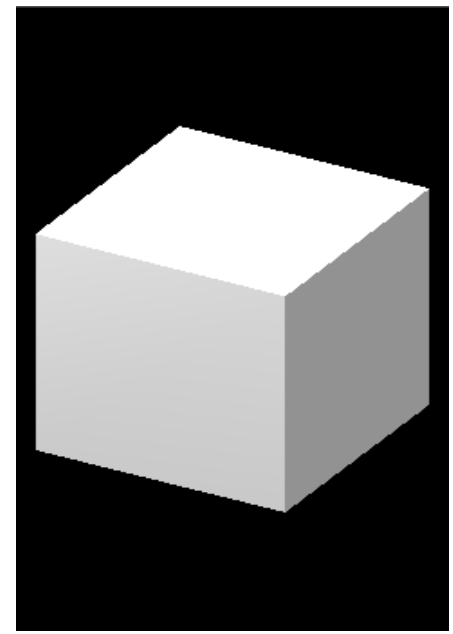
In this case the viewing volume is a parallelepiped. All objects with the same dimension appear the same size, regardless of whether they are near or far from the camera.



ortho(left, right, bottom, top, near, far)
ortho(0, width, 0, height, -10, 10) (default)

```
void setup() {
  size(240, 400, A3D);
  noStroke();
  fill(204);
}

void draw() {
  background(0);
  lights();

  if(mousePressed) {
    float fov = PI/3.0;
    float cameraZ = (height/2.0) / tan(PI * fov / 360.0);
    perspective(fov, float(width)/float(height),
                cameraZ/2.0, cameraZ*2.0);
  } else {
    ortho(-width/2, width/2, -height/2, height/2, -10, 10);
  }

  translate(width/2, height/2, 0);
  rotateX(-PI/6);
  rotateY(PI/3);
  box(160);
}
```

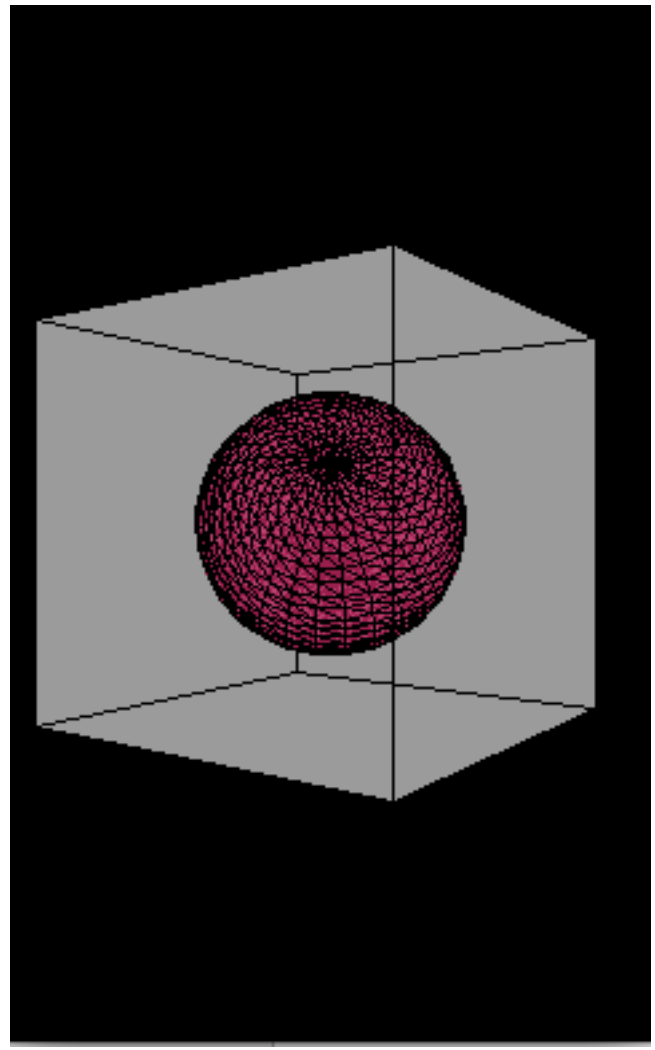




10. Creating 3D objects

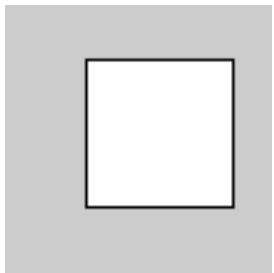
A3D provides some functions for drawing predefined 3D primitives:
sphere(r), box(w, h, d)

```
void setup() {
  size(240, 400, A3D);
  stroke(0);
}

void draw() {
  background(0);
  translate(width/2,height/2,0);

  fill(200, 200);
  pushMatrix();
  rotateY(frameCount*PI/185);
  box(150, 150, 150);
  popMatrix();

  fill(200, 40, 100, 200);
  pushMatrix();
  rotateX(-frameCount*PI/200);
  sphere(50);
  popMatrix();
}
```



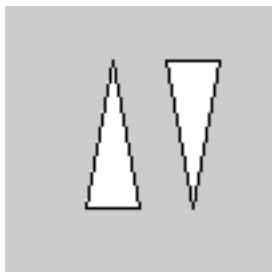
beginShape()/endShape()

1. The beginShape()/endShape() functions allow us to create complex objects by specifying the vertices and their connectivity (and optionally the normals and textures coordinates for each vertex)
2. This functionality is already present in A2D, with the difference that in A3D we can specify vertices with z coordinates.



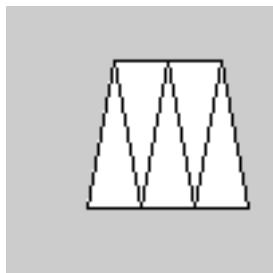
```
beginShape();
vertex(30, 20, 0);
vertex(85, 20, 0);
vertex(85, 75, 0);
vertex(30, 75, 0);
endShape(CLOSE);
```

Closed polygon



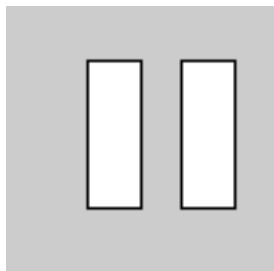
```
beginShape(TRIANGLES);
vertex(30, 75, 0);
vertex(40, 20, 0);
vertex(50, 75, 0);
vertex(60, 20, 0);
vertex(70, 75, 0);
vertex(80, 20, 0);
endShape();
```

Individual triangles



```
beginShape(TRIANGLE_STRIP);
vertex(30, 75, 0);
vertex(40, 20, 0);
vertex(50, 75, 0);
vertex(60, 20, 0);
vertex(70, 75, 0);
vertex(80, 20, 0);
vertex(90, 75, 0);
endShape();
```

Triangle strip



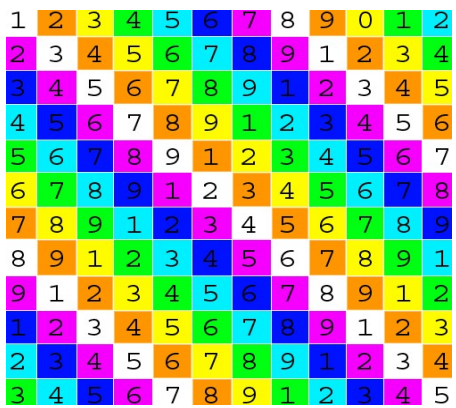
```
beginShape(QUADS);
vertex(30, 20, 0);
vertex(30, 75, 0);
vertex(50, 75, 0);
vertex(50, 20, 0);
vertex(65, 20, 0);
vertex(65, 75, 0);
vertex(85, 75, 0);
vertex(85, 20, 0);
endShape();
```

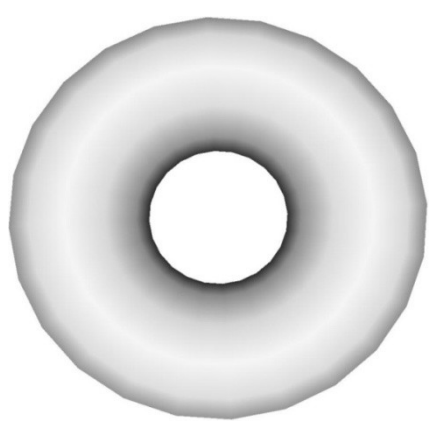
Individual quads

Check the Processing reference for more details:
http://processing.org/reference/beginShape_.html

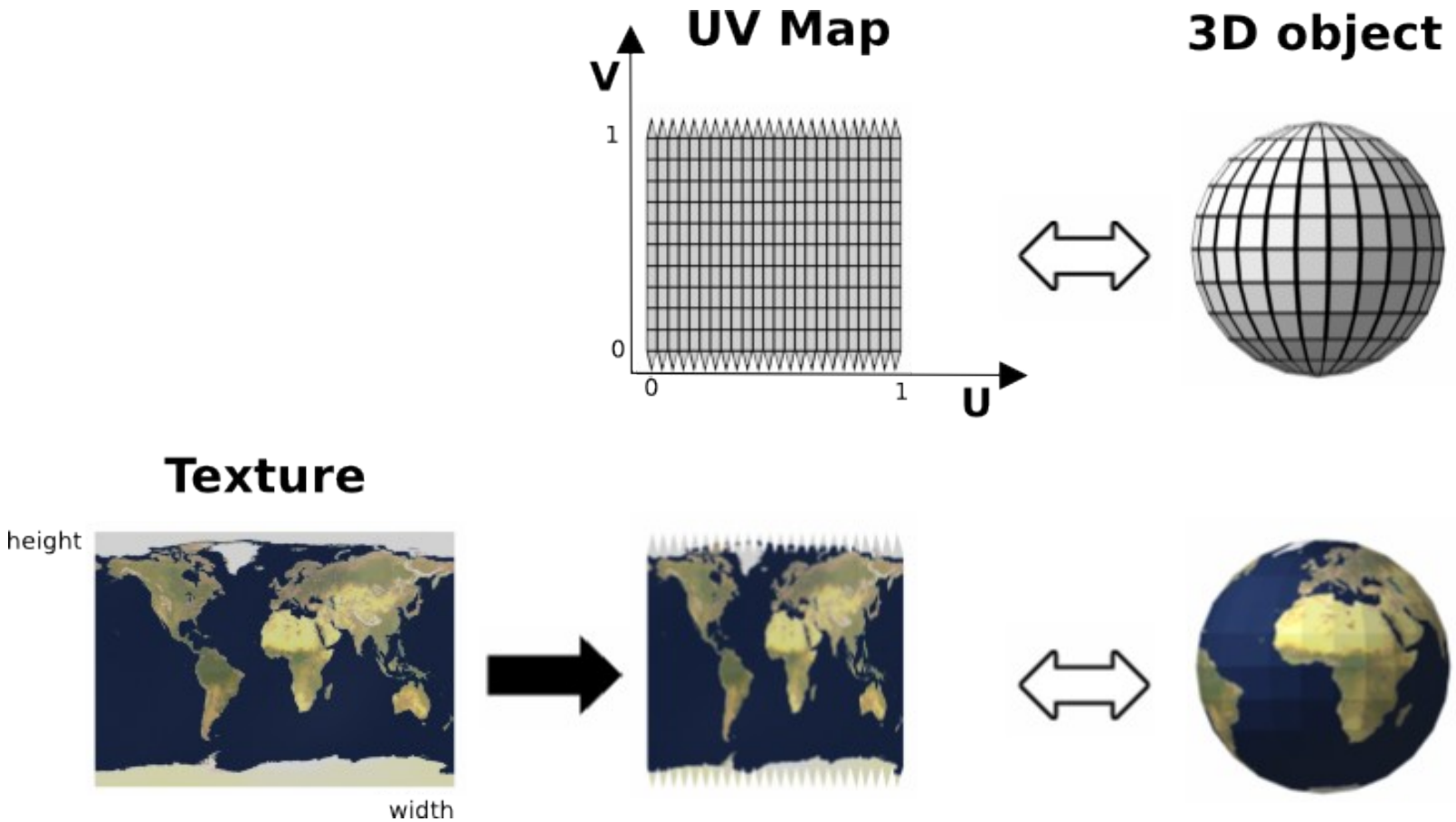
Texturing

Texturing is an important technique in computer graphics consisting in using an image to “wrap” a 3D object in order to simulate a specific material, realistic "skin", illumination effects, etc.





Basic texture mapping:



Adapted from wikipedia.org, UV mapping:
http://en.wikipedia.org/wiki/UV_mapping

Texture mapping becomes a very complex problem when we need to texture complicated tridimensional shapes (organic forms).

Finding the correct mapping from 2D image to 3D shape requires mathematical techniques that takes into account edges, folds, etc.

Image from: Ptex: Per-Face Texture Mapping for Production Rendering, by Brent Burley and Dylan Lacewell

Simple shape texturing

Objects created with beginShape()/endShape() can be textured using any image loaded into Processing with the loadImage() function or created procedurally by manipulating the pixels individually.

```
PImage img;

void setup() {
  size(240, 240, A3D);
  img = loadImage("beach.jpg");
  textureMode(NORMAL);
}

void draw() {
  background(0);
  beginShape(QUADS);
  texture(img);
  vertex(0, 0, 0, 0, 0);
  vertex(width, 0, 0, 1, 0);
  vertex(width, height, 0, 1, 1);
  vertex(0, height, 0, 0, 1);
  endShape();
}
```

The texture mode can be NORMAL or IMAGE

Depending on the texture mode, we use normalized UV values or relative to the image resolution.

beginShape/endShape in A3D supports setting more than one texture for different parts of the shape:

```
PImage img1, img2;

void setup() {
  size(240, 240, A3D);
  img1 = loadImage("beach.jpg");
  img2 = loadImage("peebles.jpg");
  textureMode(NORMAL);
  noStroke();
}

void draw() {
  background(0);
  beginShape(TRIANGLES);
  texture(img1);
  vertex(0, 0, 0, 0, 0);
  vertex(width, 0, 0, 1, 0);
  vertex(0, height, 0, 0, 1);
  texture(img2);
  vertex(width, 0, 0, 1, 0);
  vertex(width, height, 0, 1, 1);
  vertex(0, height, 0, 0, 1);
  endShape();
}
```

Lighting

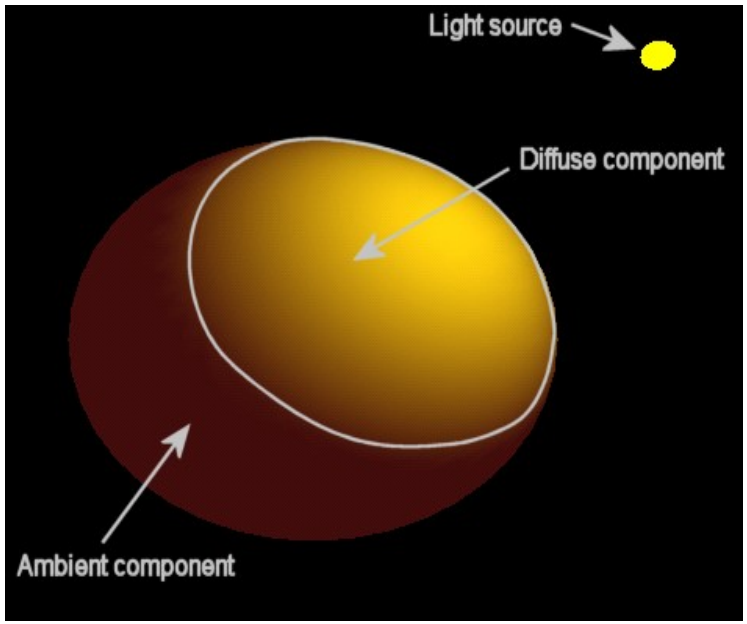
1. A3D offers a local illumination model based on OpenGL's model.
2. It is a simple real-time illumination model, where each light source has 4 components: ambient + diffuse + specular + emissive = total
3. This model doesn't allow the creation of shadows
4. We can define up to 8 light sources.
5. Proper lighting calculations require to specify the normals of an object

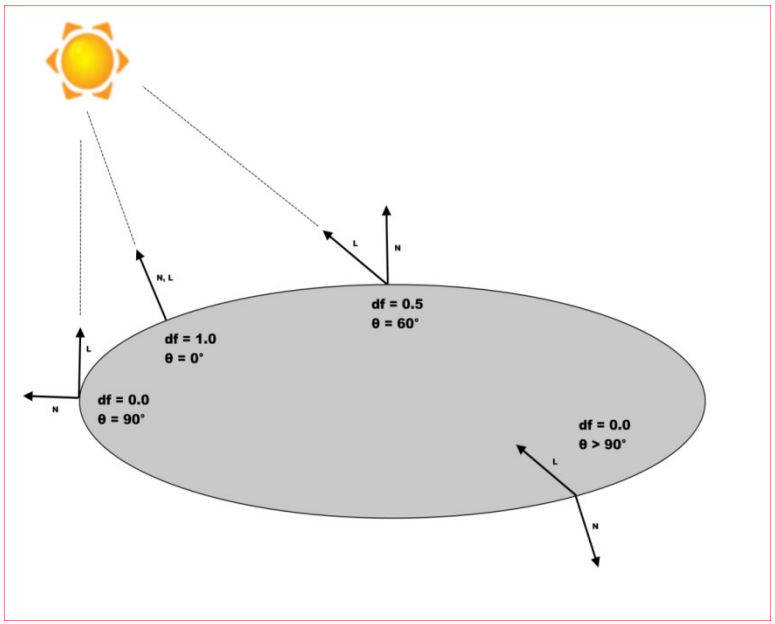
Ambient **Diffuse** **Specular**

From http://www.falloutsoftware.com/tutorials/gl/gl8.htm

Some more good resources about lights in OpenGL:
http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson6.php
http://jerome.jouvie.free.fr/OpenGl/Tutorials/Tutorial12.php - Tutorial15.php
http://www.sjbaker.org/steve/omniv/opengl_lighting.html

In diffuse lighting, the angle between the normal of the object and the direction to the light source determines the intensity of the illumination:





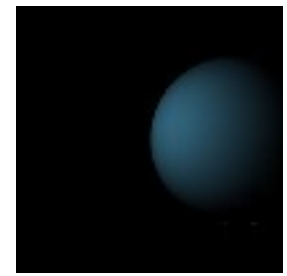
From iPhone 3D programming, by Philip Rideout.
http://iphone-3d-programming.labs.oreilly.com/ch04.html
http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson6.php

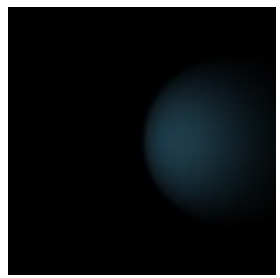
Light types in A3D

Ambient: Ambient light doesn't come from a specific direction, the rays have light have bounced around so much that objects are evenly lit from all sides. Ambient lights are almost always used in combination with other types of lights.
ambientLight(v1, v2, v3, x, y, z)
v1, v2, v3: rgb color of the light
x, y, z position:

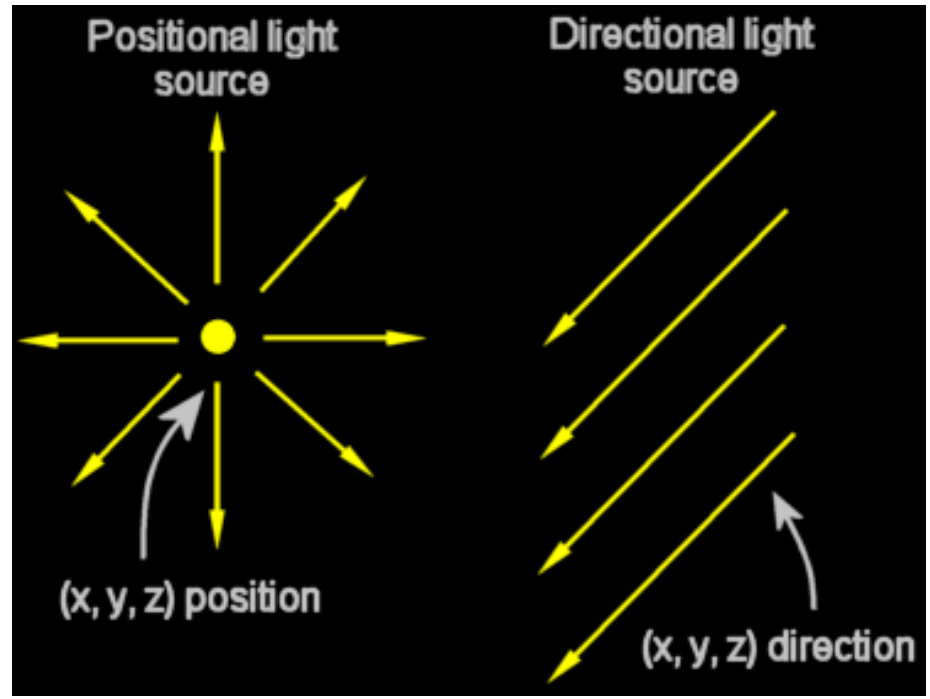
Directional: Directional light comes from one direction and is stronger when hitting a surface squarely and weaker if it hits at a a gentle angle. After hitting a surface, a directional lights scatters in all directions.
directionalLight(v1, v2, v3, nx, ny, nz)
v1, v2, v3: rgb color of the light
nx, ny, and nz the direction the light is facing.

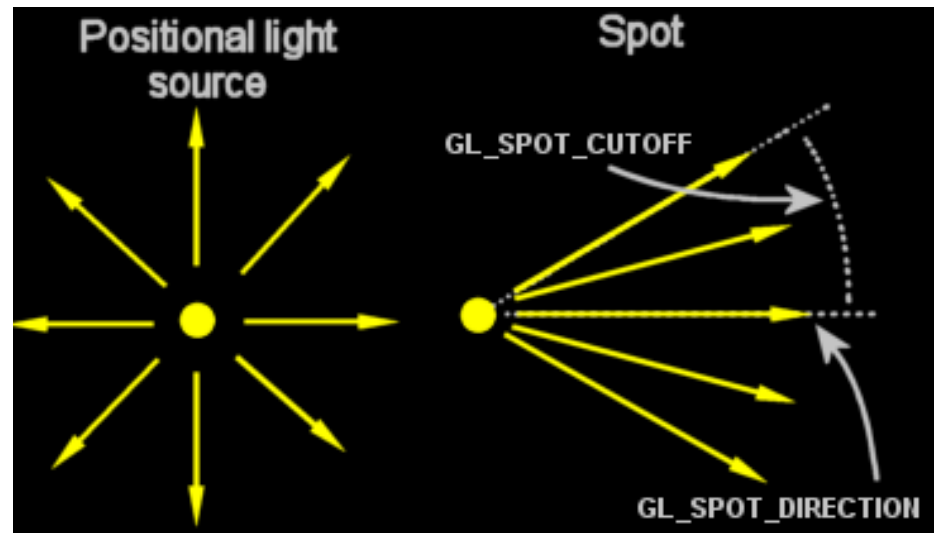
Point: Point light irradiates from a specific position.
pointLight(v1, v2, v3, x, y, z)
v1, v2, v3: rgb color of the light
x, y, z position:





Spot: A spot light emits lights into an emission cone by restricting the emission area of the light source.
spotLight(v1, v2, v3, x, y, z, nx, ny, nz, angle, concentration)
v1, v2, v3: rgb color of the light
x, y, z position:
nx, ny, nz specify the direction or light
angle float: angle the spotlight cone
concentration: exponent determining the center bias of the cone

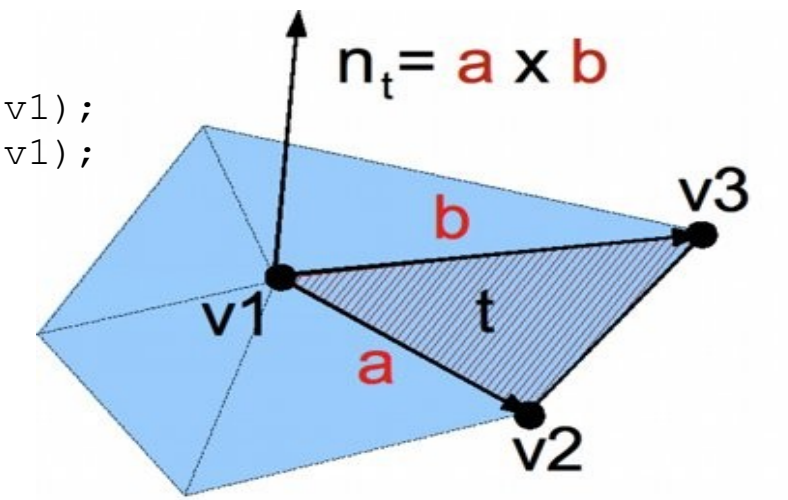




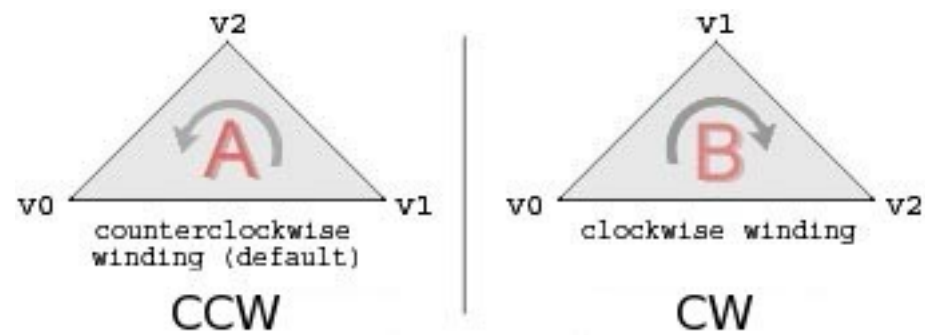
http://jerome.jouvie.free.fr/OpenGl/Lessons/Lesson6.php

Normals: each vertex needs to have a normal defined so the light calculations can be performed correctly

```
PVector a = PVector.sub(v2, v1);
PVector b = PVector.sub(v3, v1);
PVector n = a.cross(b);
normal(n.x, n.y, n.z);

vertex(v1.x, v1.y, v1.z);
vertex(v2.x, v2.y, v2.z);
vertex(v3.x, v3.y, v3.z);
```



Polygon winding: The ordering of the vertices that define a face determine which side is inside and which one is outside. Processing uses CCW ordering of the vertices, and the normals we provide to it must be consistent with this.



11. 3D Text

Text in A3D works exactly the same as in A2D:

1. load/create fonts with loadFont/createFont
2. set current font with textFont
3. write text using the text() function

```
PFont fontA;
void setup()  {
  size(240, 400, A3D);
  background(102);
  String[] fonts = PFont.list();
  fontA = createFont(fonts[0], 32);
  textFont(fontA, 32);
}

void draw() {
  fill(0);
  text("An", 10, 60);
  fill(51);
  text("droid", 10, 95);
  fill(204);
  text("in", 10, 130);
  fill(255);
  text("A3D", 10, 165);
}
```

1. The main addition in A3D is that text can be manipulated in three dimensions.
2. Each string of text we print to the screen with text() is contained in a rectangle that we can rotate, translate, scale, etc.
3. The rendering of text is also very efficient because is accelerated by the GPU (A3D internally uses OpenGL textures to store the font characters)

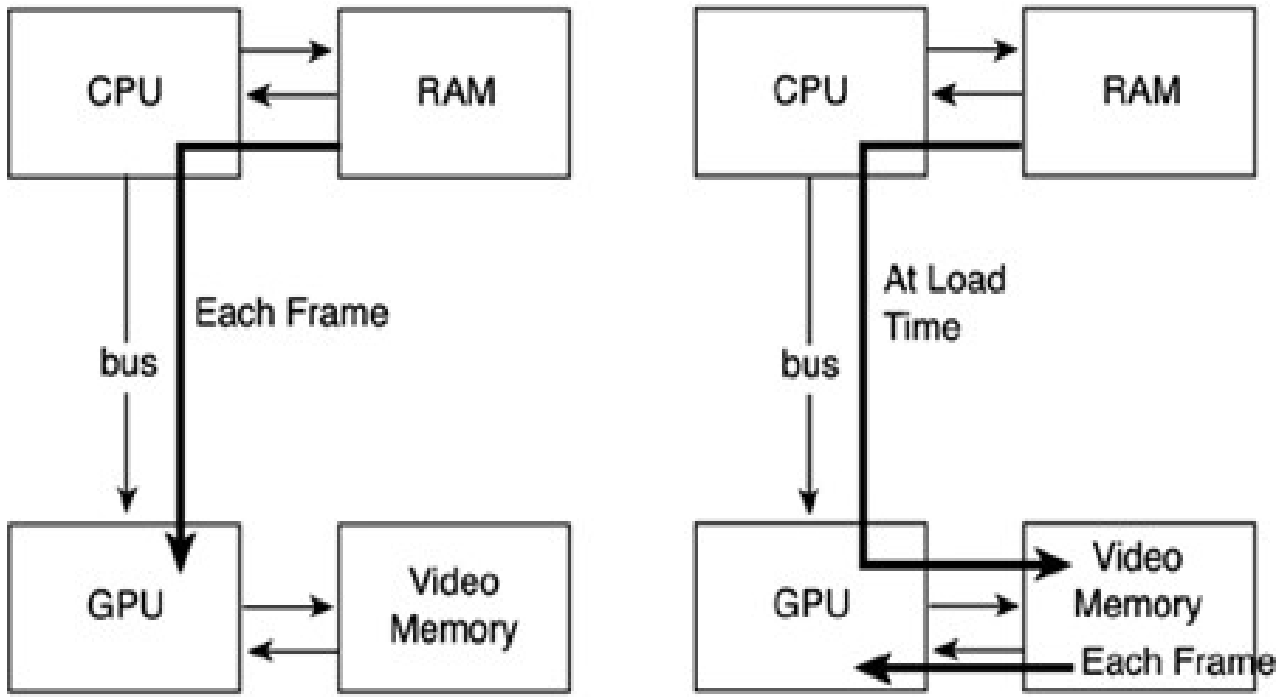
```
fill(0);
pushMatrix();
translate(rPos,10+25);
char k;
for(int i = 0;i < buff.length(); i++) {
  k = buff.charAt(i);
  translate(-textWidth(k),0);
  rotateY(-textWidth(k)/70.0);
  rotateX(textWidth(k)/70.0);
  scale(1.1);
  text(k,0,0);
}
popMatrix();
```

```
PFont font;
char[] sentence = { 'S', 'p', 'A' , 'O', '5', 'Q',
                    'S', 'p', 'A' , 'O', '5', 'Q',
                    'S', 'p', 'A' , 'O', '5', 'Q',
                    'S', 'p', 'A' , 'O', '5', 'Q' };

void setup() {
  size(240, 400, P3D);
  font = loadFont("Ziggurat-HTF-Black-32.vlw");
  textFont(font, 32);
}

void draw() {
  background(0);

  translate(width/2, height/2, 0);

  for (int i = 0; i < 24; i++) {
    rotateY(TWO_PI / 24 + frameCount * PI/5000);
    pushMatrix();
    translate(100, 0, 0);
    //box(10, 50, 10);
    text(sentence[i], 0, 0);
    popMatrix();
  }
}
```

12 Models: the Pshape3D class

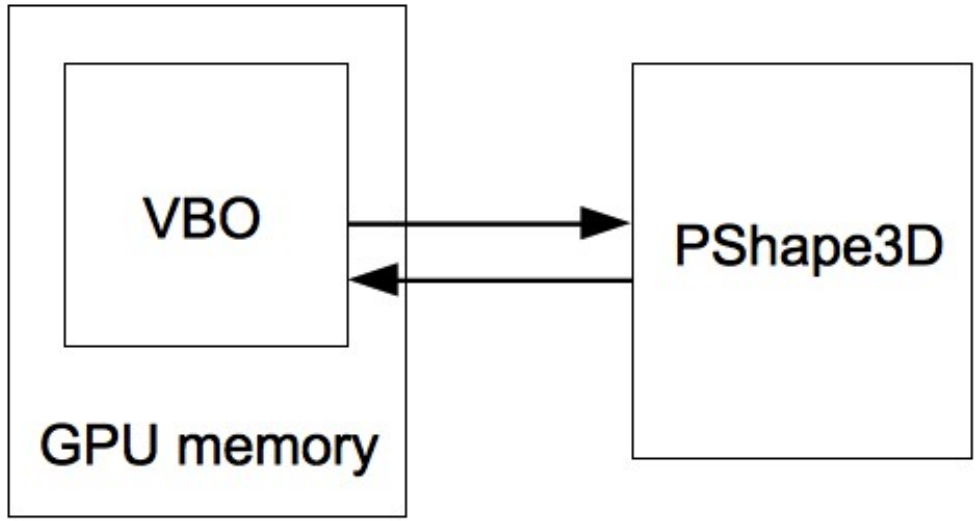
Vertex Buffer Objects

1. Normally, the data that defines a 3D object (vertices, colors, normals, texture coordinates) are sent to the GPU at every frame.
2. The current GPUs in mobile devices have limited bandwidth, so data transfers can be slow.
3. If the geometry doesn't change (often) we can use Vertex Buffer Objects.
4. A Vertex Buffer Object is a piece of GPU memory where we can upload the data defining an object (vertices, colors, etc.)
5. The upload (slow) occurs only once, and once the VBO is stored in GPU memory, we can draw it without uploading it again.
6. This is similar to the concept of Textures (upload once, use multiple times).



For a good tutorial about VBOs, see this page:
http://www.songho.ca/opengl/gl_vbo.html

The PShape3D class in A3D encapsulates VBOs



1. The class Pshape3D in A3D encapsulates a VBO and provides a simple way to create and handle VBO data, including updates, data fetches, texturing, loading from OBJ files, etc.
2. PShape3D has to be created with the total number of vertices know beforehand. Resizing is possible, but slow.
3. How vertices are interpreted depends on the geometry type specified at creation (POINT, TRIANGLES, etc), in a similar way to beginShape()/endShape()
4. Vertices in a PShape3D can be divided in groups, to facilitate normal and color assignment, texturing and creation of complex shapes with multiple geometry types (line, triangles, etc).

Manual creation of Pshape3D models

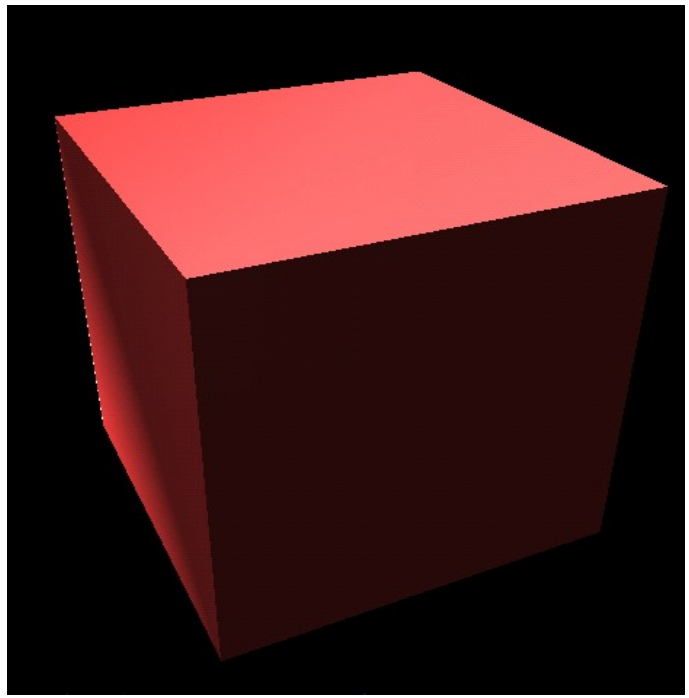
1. A PShape3D can be created by specifying each vertex and associated data (normal, color, etc) manually.
2. Remember that the normal specification must be consistent with the CCW vertex ordering.

Creation

```
cube = createShape(36, TRIANGLES);

cube.beginUpdate(VERTICES);
cube.setVertex(0, -100, +100, -100);
cube.setVertex(1, -100, -100, -100);
...
cube.endUpdate();

cube.beginUpdate(COLORS);
cube.setVertex(0, color(200, 50, 50, 150));
cube.setVertex(1, color(200, 50, 50, 150));
...
cube.endUpdate();

cube.beginUpdate(NORMALS);
cube.setVertex(0, 0, 0, -1);
cube.setVertex(1, 0, 0, -1);
...
cube.endUpdate();
```

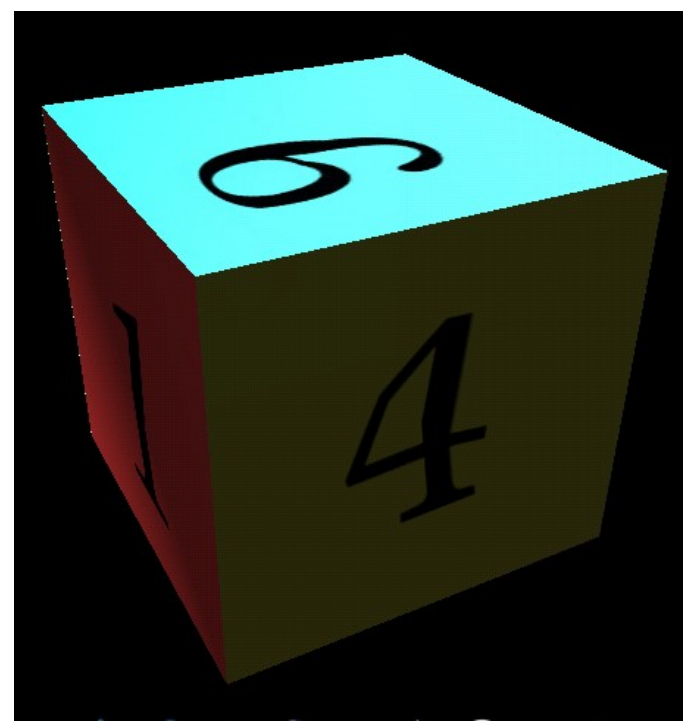
Drawing

```
translate(width/2, height/2, 0);
shape(cube);
```



1. A PShape3D can be textured with one ore more images.
2. The vertices can be organized in groups, which allows to assign a different texture to each group.
3. Groups also facilitate the assignment of colors and normals.

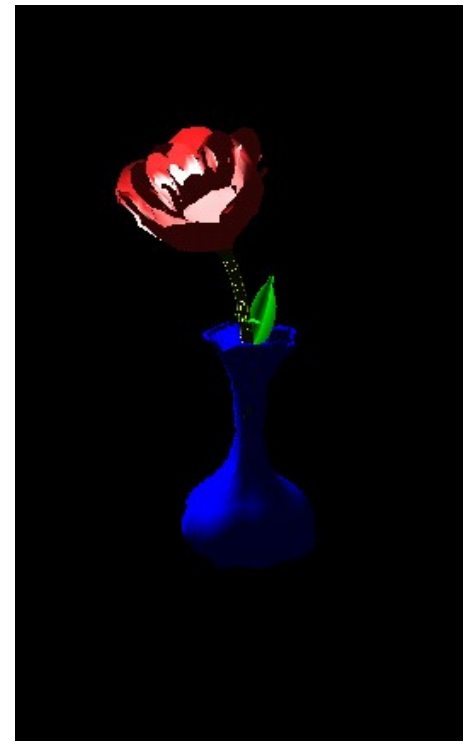
```
cube.beginUpdate(VERTICES);
cube.setGroup(0);
cube.setVertex(0, -100, +100, -100);
cube.setVertex(1, -100, -100, -100);
...
cube.setGroup(1);
cube.setVertex(6, +100, -100, +100);
cube.setVertex(7, +100, +100, +100);
...
cube.endUpdate();

cube.setGroupColor(0, color(200, 50, 50,
150));
cube.setGroupColor(1, color(200, 50, 50,
150));
...

cube.setGroupNormal(0, 0, 0, -1);
cube.setGroupNormal(1, +1, 0, 0);
...
cube.setGroupTexture(0, face0);
cube.setGroupTexture(1, face1);
...
```



OBJ loading

1. The OBJ format is a text-based data format to store 3D geometries. There is an associated MTL format for materials definitions.
2. It is supported by many tools for 3D modeling (Blender, Google Sketchup, Maya, 3D Studio). For more info: http://en.wikipedia.org/wiki/Obj
3. A3D supports loading OBJ files into PShape3D objects with the loadShape() function.
4. Depending the extension of the file passed to loadShape (.svg or .obj) Processing will attempt to interpret the file as either SVG or OBJ.
5. The styles active at the time of loading the shape are used to generate the geometry.

```
PShape object;
float rotX;
float rotY;

void setup() {
  size(480, 800, A3D);
  noStroke();
  object = loadShape("rose+vase.obj");
}

void draw() {
  background(0);
  ambient(250, 250, 250);
  pointLight(255, 255, 255, 0, 0, 200);
  translate(width/2, height/2, 400);
  rotateX(rotY);
  rotateY(rotX);
  shape(object);
}
```



Copying SVG shapes into PShape3D

Once we load an SVG file into a PShapeSVG object, we can copy into a PShape3D for increased performance:

```
PShape bot;
PShape3D bot3D;

public void setup() {
  size(480, 800, A3D);
  bot = loadShape("bot.svg");
  bot3D = createShape(bot);
}

public void draw() {
  background(255);

  shape(bot3D, mouseX, mouseY, 100, 100);
}
```

Shape recording

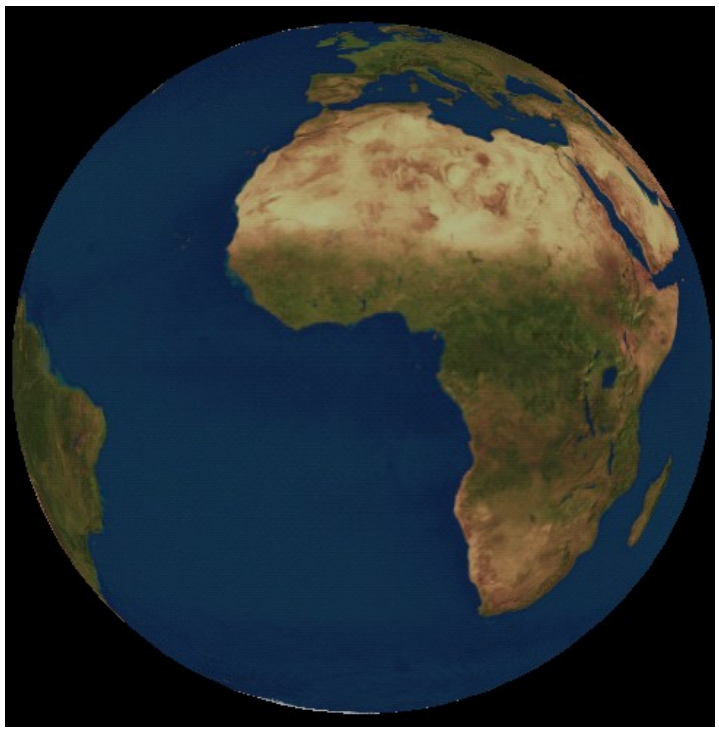
1. Shape recording into a PShape3D object is a feature that can greatly increase the performance of sketches that use complex geometries.
2. The basic idea of shape recording is to save the result of standard Processing drawing calls into a Pshape3D object.
3. There are two ways of using shape recording: one is saving a single shape with beginShapeRecorder/endShapeRecorder, and the second allows saving multiple shapes with beginShapesRecorder/endShapesRecorder

```
Pshape recShape;

void setup() {
  size(480, 800, A3D);

  beginShapeRecorder(QUADS);
  vertex(50, 50);
  vertex(width/2, 50);
  vertex(width/2, height/2);
  vertex(50, height/2);
  recShape = endShapeRecorder();
  ...
}

void draw() {
  ...
  shape(recShape3D);
  ...
}
```

```
PShape3D objects;

void setup() {
  size(480, 800, A3D);
  beginShapesRecorder();
  box(1, 1, 1);
  rect(0, 0, 1, 1);
  ...
  objects = endShapesRecorder();
  ...
}

void draw() {
  ...
  shape(objects);
  ...
}
```

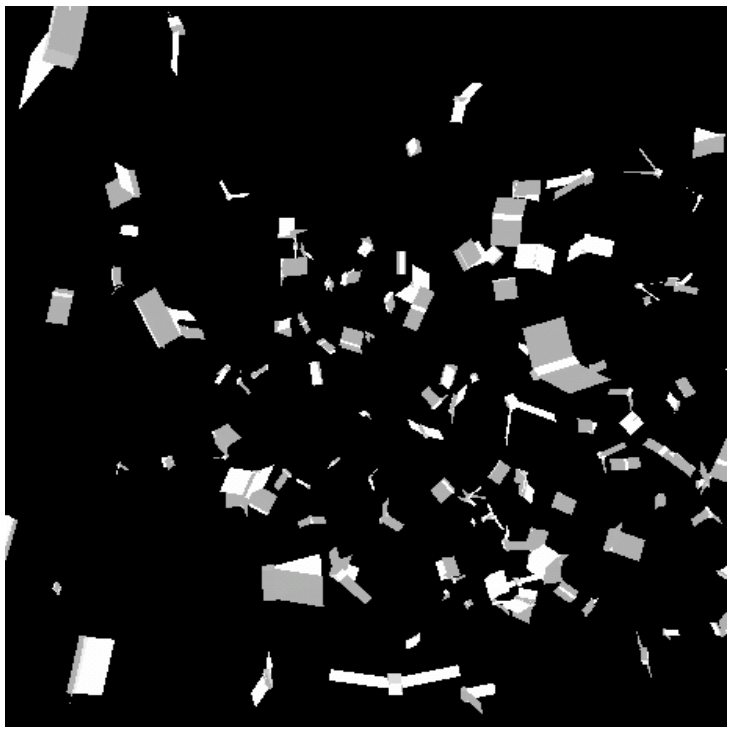
The performance gains of using shape recording are quite substantial.
It usually increases the rendering framerate by 100% or more.

Textured sphere



Without shape recording: 19fps
With shape recording: 55fps

Birds flock



Without shape recording: 7fps
With shape recording: 18fps

Particle systems

PShape3D allows to create models with the POINT_SPRITES geometry type.
With this type, each vertex is treated as a textured point sprite.
At least one texture must be attached to the model, in order to texture the sprites.
More than sprite texture can be attached, by dividing the vertices in groups.
The position and color of the vertices can be updated in the draw loop in order to simulate motion (we have to create the shape as DYNAMIC).

```
particles = createShape(1000, POINT_SPRITES, DYNAMIC);
particles.beginUpdate(VERTICES);
for (int i =0; i < particles.getNumVertices(); i++) {
  float x = random(-30, 30);
  float y = random(-30, 30);
  float z = random(-30, 30);
  particles.setVertex(i, x, y, z);
}
particles.endUpdate();
sprite = loadImage("particle.png");
particles.setTexture(sprite);
```

Creation/initialization

```
particles.beginUpdate(VERTICES);
for (int i =0; i < particles.getNumVertices(); i++) {
    float[] p = particles.getVertex(i);
    p[0] += random(-1, 1);
    p[1] += random(-1, 1);
    p[2] += random(-1, 1);
    particles.setVertex(i, p);
  }
particles.endUpdate();
```

Dynamic update

Wrapping up...

Lets look an example where we do:

1. offscreen rendering to texture
2. particle system (rendered to offscreen surface)
3. dynamic texturing of a 3D shape using the result of the offscreen drawing

Links

Very good Android tutorial: http://www.vogella.de/articles/Android/article.html

Official google resources: http://developer.android.com/index.html
SDK: http://developer.android.com/sdk/index.html
Guide: http://developer.android.com/guide/index.html
OpenGL: http://developer.android.com/guide/topics/graphics/opengl.html
Mailing list: http://groups.google.com/group/android-developers

Developers forums: http://www.anddev.org/
Book: http://andbook.anddev.org/

Cyanogenmod project: http://www.cyanogenmod.com/

GLES benchmarks: http://www.glbenchmark.com/result.jsp

Min3D (framework 3D): http://code.google.com/p/min3d/

Developer's devices: http://www.hardkernel.com/

AppInventor: http://www.appinventor.org/

Processing resources: http://processing.org/
Processing.Android forum: http://forum.processing.org/android-processing
Processing.Android forum: http://wiki.processing.org/w/Android