3D Mobile Games

In this lesson...

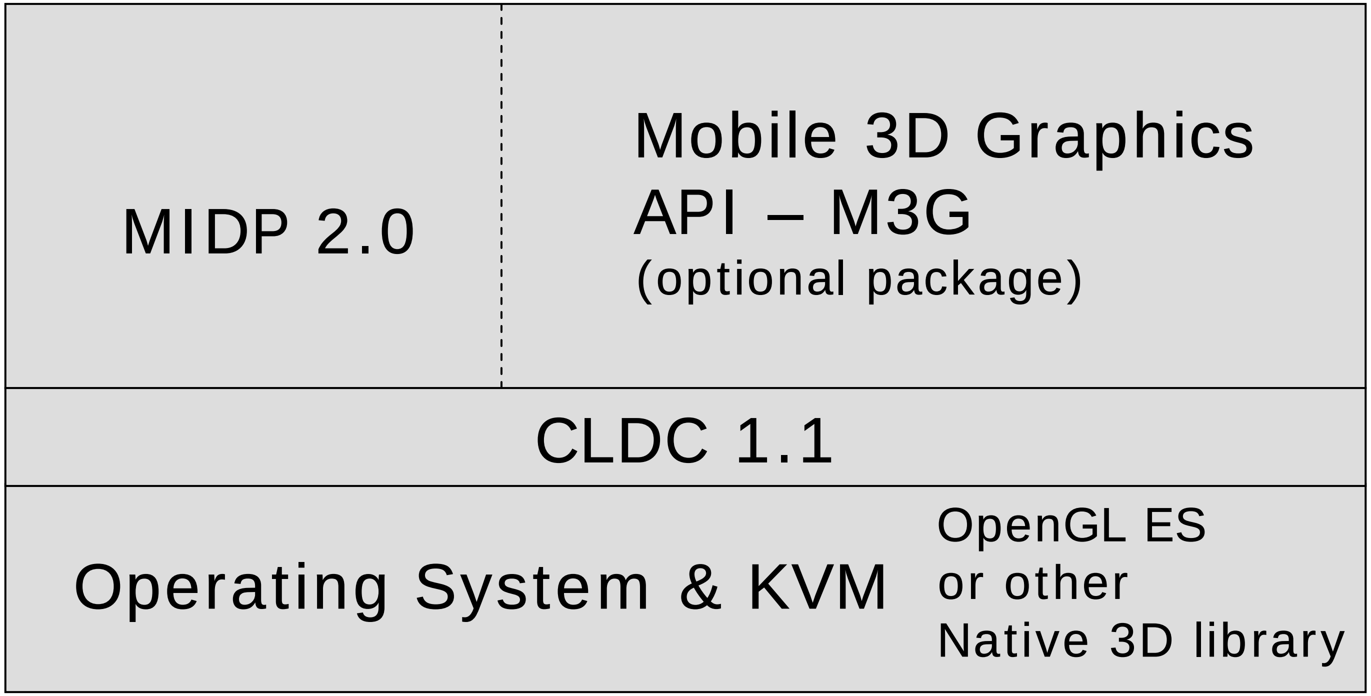
- 3D Mobile Games
  - ✶Immediate mode
  - ✶Retained mode

3D Mobile Games

In this lesson…

- 3D Mobile Games
    - ✶Retained mode
    - ✶Immediate Mode

Mobile 3D Graphics API



MIDP 2.0

Mobile 3D Graphics API – M3G
(optional package)

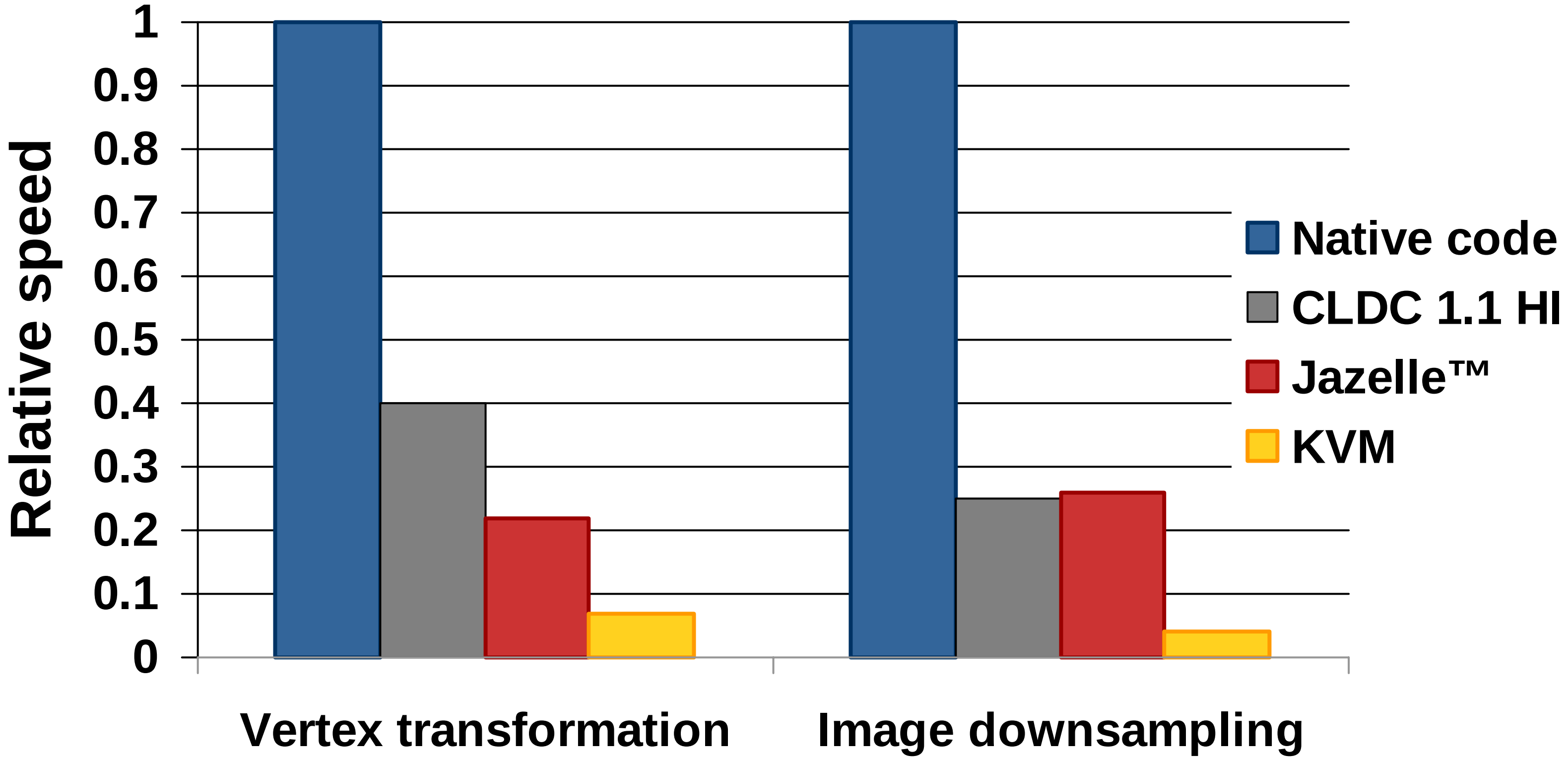
CLDC 1.1

Operating System & KVM

OpenGL ES
or other
Native 3D library

Overcome the performance barrier

Native (C/C++) vs. Java on mobiles



Benchmarked on an ARM9 processor

Source: nokia.com

Why a new standard?

- OpenGL (ES) is too low-level
  - Lots of Java code needed for simple things
- Java 3D™ API is too bloated
  - A hundred times larger than M3G
  - Does not fit together with MIDP
  - Tried and failed, but…
- Now knew what we wanted!
  - Basic Java 3D™ ideas: nodes, scene graph
  - Add file format, keyframe animation
  - Remain compatible with OpenGL ES

Graphics3D

- Contains global state
  - Target surface, viewport, depth buffer
  - Camera, light sources
  - Rendering quality hints

- Each renderable object has its own local state
  - Geometry and appearance (material, textures, etc.)
  - Transformation relative to parent or world

Graphics3D: Rendering modes

- Retained mode
  - Render a scene graph, rooted by the World
  - Take the Camera and Lights from the World
- Immediate mode
  - Render a branch or an individual node at a time
  - Explicitly give the Camera and Lights to Graphics3D

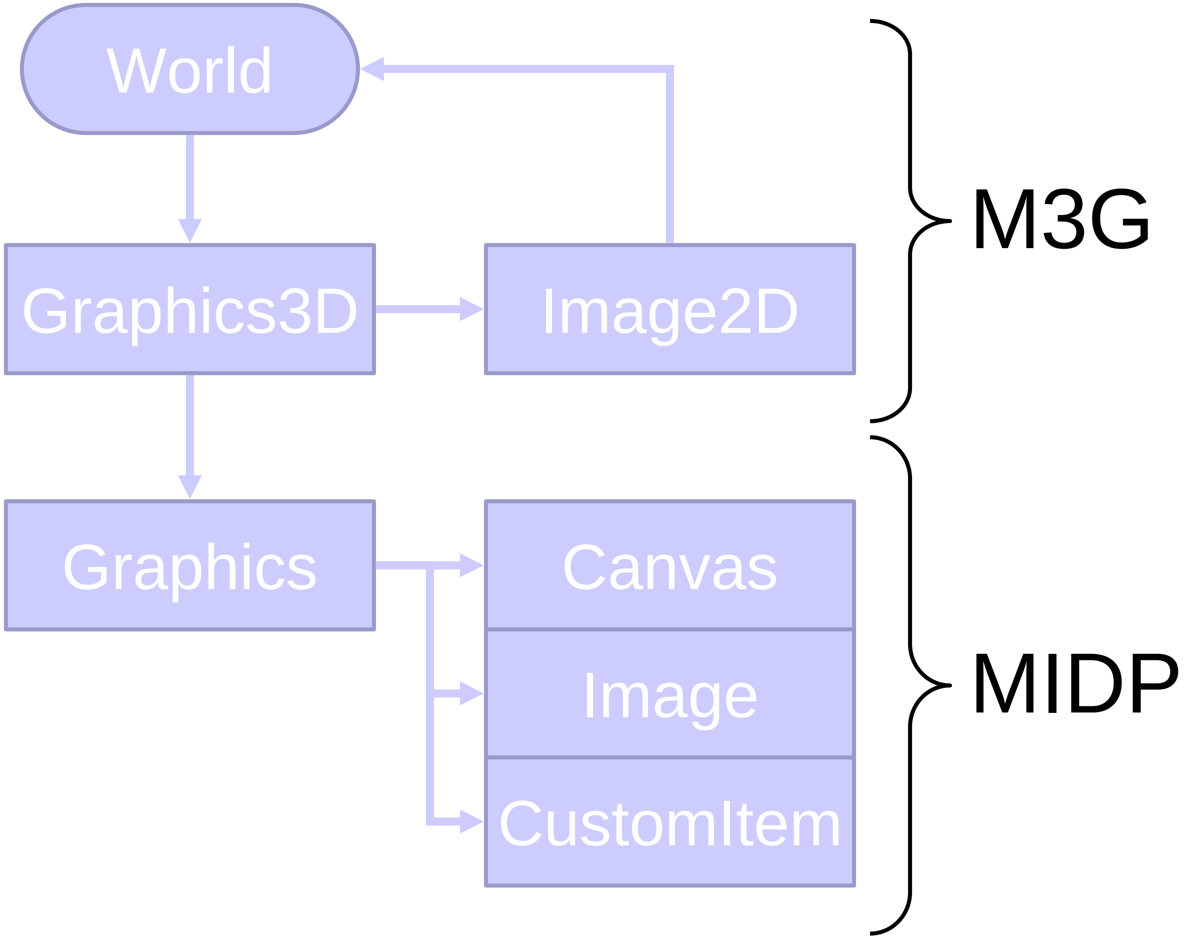
Graphics3D

- ➢ Using Graphics3D
  - ✶ Bind a target to it
  - ✶ Render it
  - ✶ release the target

```
Graphics3D g3d = Graphics3D.getInstance();
World w = (World) Loader.load("/file.m3g")[0];
void paint(Graphics g) {
  myGraphics3D.bindTarget(g);
  myGraphics3D.render(world);
  myGraphics3D.releaseTarget();
}
```

- ➢ Note: Everything is synchronous
  - ✶ A method returns only when it’s done
  - ✶ No separate thread for renderer or loader

Graphics3D: Rendering targets



A simplified animation player

```
import javax.microedition.midlet.MIDlet;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.game.GameCanvas;
import javax.microedition.m3g.*;

public class Player extends MIDlet
{
  public void pauseApp() {}

  public void destroyApp(boolean b) {}

  public void startApp() {
    PlayerCanvas canvas = new PlayerCanvas(true);
    Display.getDisplay(this).setCurrent(canvas);
    try { canvas.disp(); } catch (Exception e) {}
    notifyDestroyed();
  }
}
```

A simplified animation player

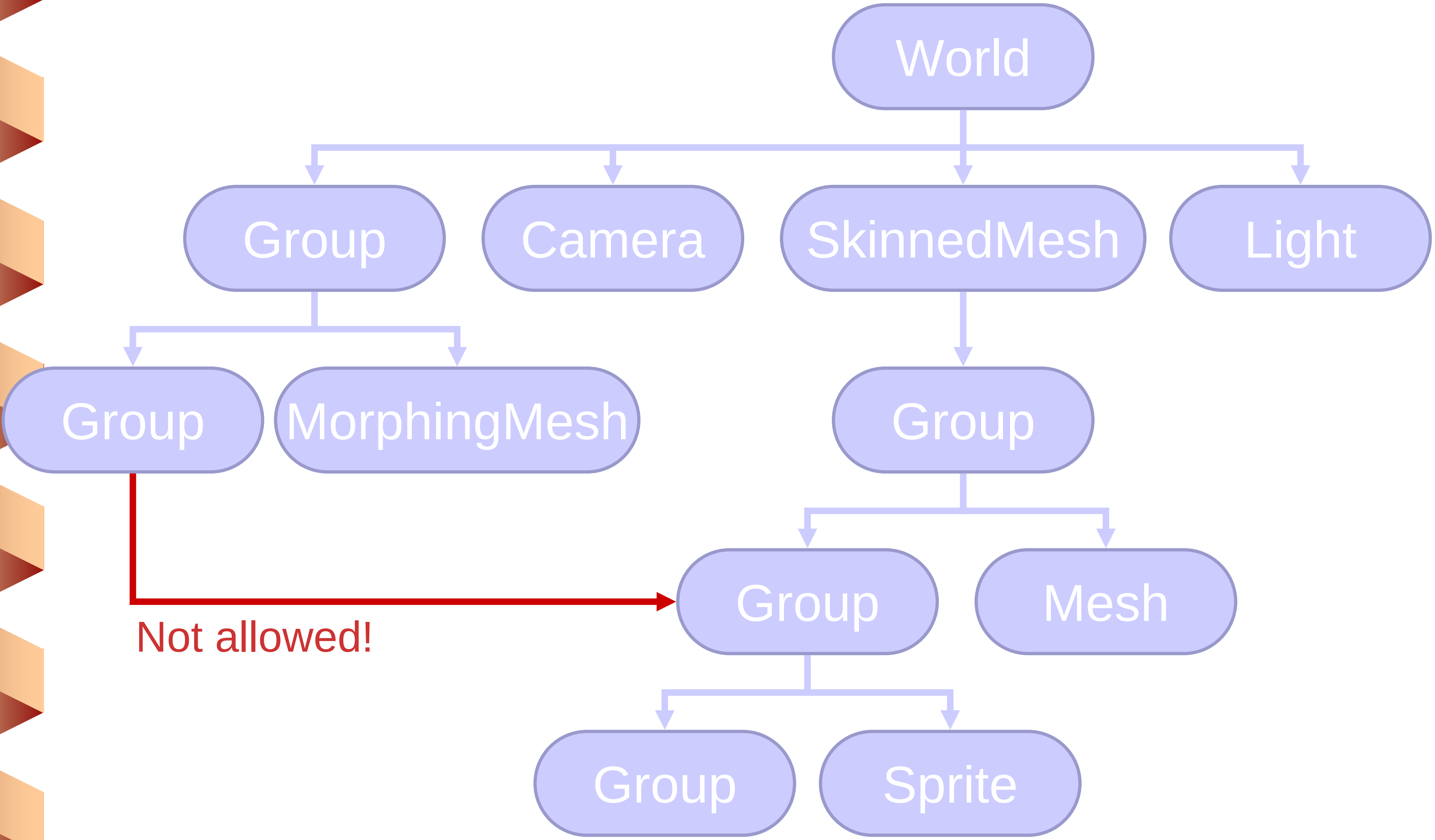
```
class PlayerCanvas extends GameCanvas {
 PlayerCanvas(boolean suppress){super(suppress);}

 public void disp() throws Exception {
  Graphics3D g3d = Graphics3D.getInstance();
  World w = (World) Loader.load("/skaterboy.m3g")[0];
  long start, elapsed, time = 0;
  while (getKeyStates() == 0) {
   start = System.currentTimeMillis();
   g3d.bindTarget(getGraphics());
   try {
    w.animate((int)time);
    g3d.render(w);
   } finally { g3d.releaseTarget(); }
   flushGraphics();
   elapsed = System.currentTimeMillis()-start;
   time += (elapsed < 100) ? 100 : (int)elapsed;
   if (elapsed < 100) Thread.sleep(100-elapsed);
  }
 }
}
```

Returns reference to object at index 0.
-> Root - World

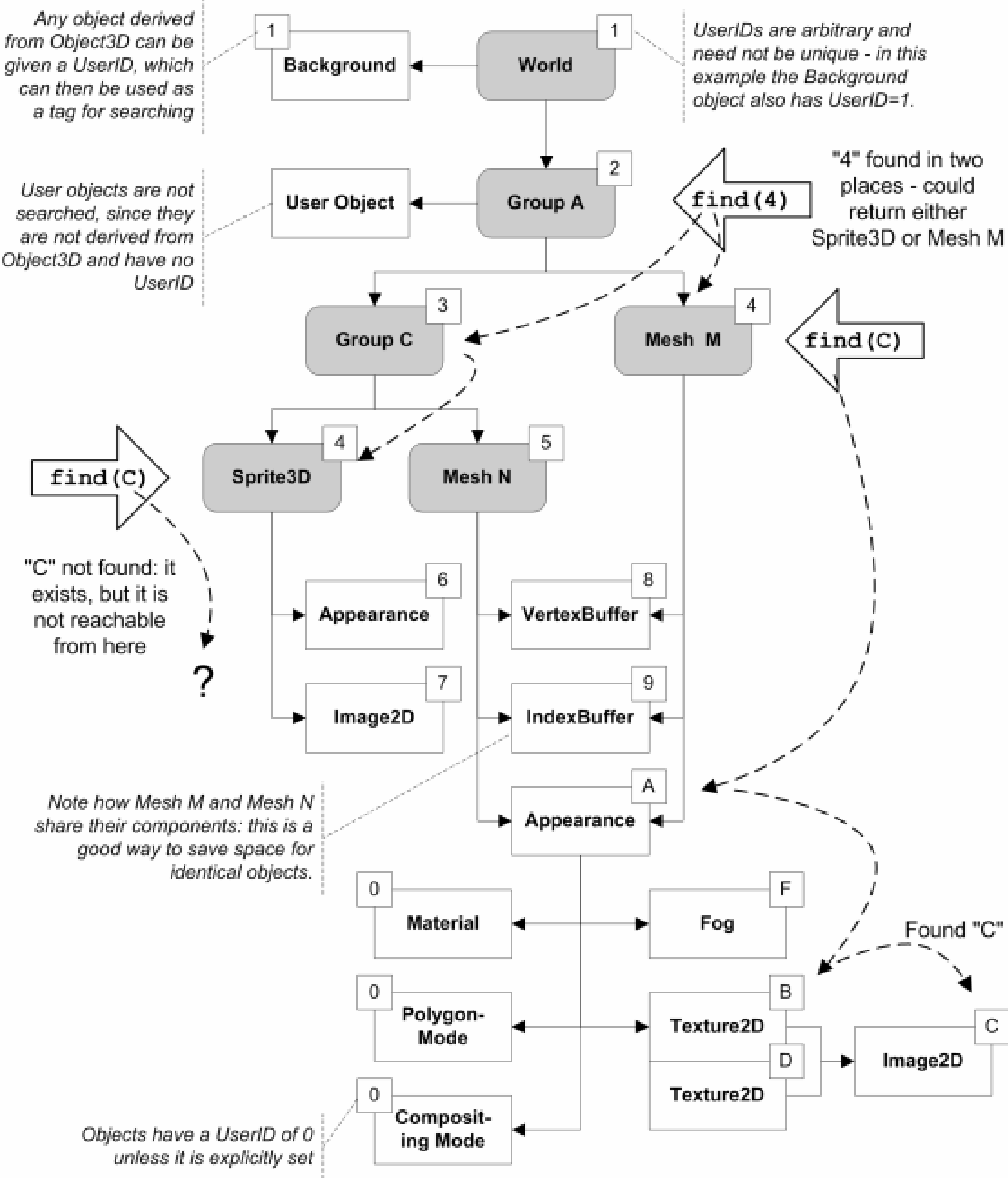
Obtaining Objects in the World

- Camera getActiveCamera();
- Backgruond getBackground();
- Every Object3D can be assigned a *user ID*, either at authoring stage or at run time with the setUserID method. User IDs are typically used to find a known object in a scene loaded from a data stream.
- Object3D find(int userID);

The scene graph



Obtaining Objects in the World



Any object derived from Object3D can be given a UserID, which can then be used as a tag for searching

UserIDs are arbitrary and need not be unique - in this example the Background object also has UserID=1.

User objects are not searched, since they are not derived from Object3D and have no UserID

"4" found in two places - could return either Sprite3D or Mesh M

"C" not found: it exists, but it is not reachable from here

Note how Mesh M and Mesh N share their components: this is a good way to save space for identical objects.

Objects have a UserID of 0 unless it is explicitly set

Found "C"

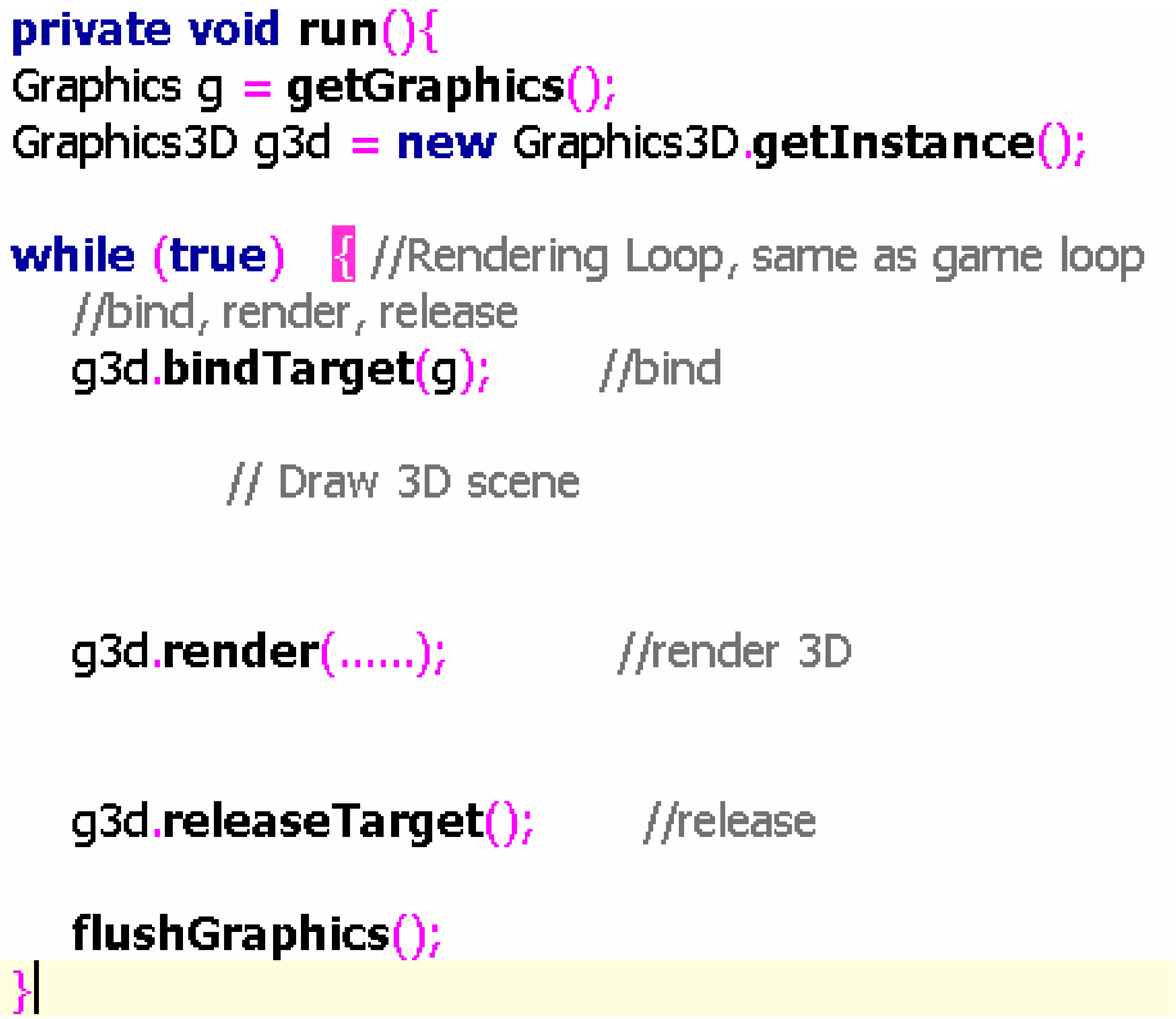
DEMO

3D Mobile Games

In this lesson…

- ➢ 3D Mobile Games
  - ✶Retained mode
  - ✶Immediate Mode

Rendering Loop



```
private void run(){
Graphics g = getGraphics();
Graphics3D g3d = new Graphics3D.getInstance();

while (true)   { //Rendering Loop, same as game loop
    //bind, render, release
    g3d.bindTarget(g);       //bind

              // Draw 3D scene


    g3d.render(......);          //render 3D


    g3d.releaseTarget();       //release

    flushGraphics();
}
```

Using the VertexArray

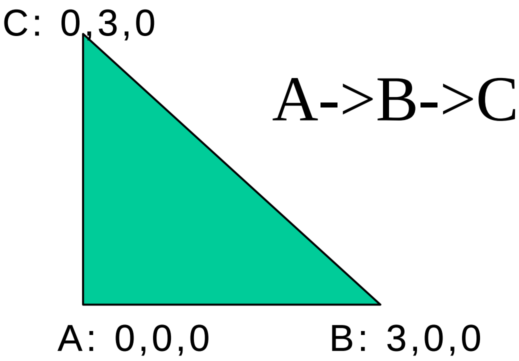
VertexArray(int numVertices, int numComponents, int componentSize)

- » numVertices - number of vertices in this VertexArray; must be [1, 65535]
- » numComponents - number of components per vertex; must be [2,3,4]
- » componentSize - number of bytes per component; must be [1, 2]

Defining the vertices of a triangle

Example 1:

```
short[] vertices = { 0, 0, 0,  3, 0, 0,   0, 3, 0};
VertexArray vertexArray = new VertexArray(vertices.length / 3, 3, 2);
vertexArray.set(0, vertices.length/3, vertices);
```



-VertexArray is an M3G class that holds an array of triplets – (x, y, z).
[also (x,y) for texture and (a,b,c,d) for colors]

-Many methods in M3G take VertexArray as an input argument.

- Vertex positions must have 3 components.
- Normal vectors must have 3 components.
- Texture coordinates must have 2 or 3 components.
- Colors must have 3 or 4 components, one byte each.

set(int firstVertex, int numVertices, byte[] values)
Copies in an array of 8-bit vertex attributes.

set(int firstVertex, int numVertices, short[] values)
Copies in an array of 16-bit vertex attributes.

Other Methods: **get**(int firstVertex, int numVertices, byte[] values),
get(int firstVertex, int numVertices, short[] values),
int **getComponentCount**(), int **getComponentType**(),
int **getVertexCount**()

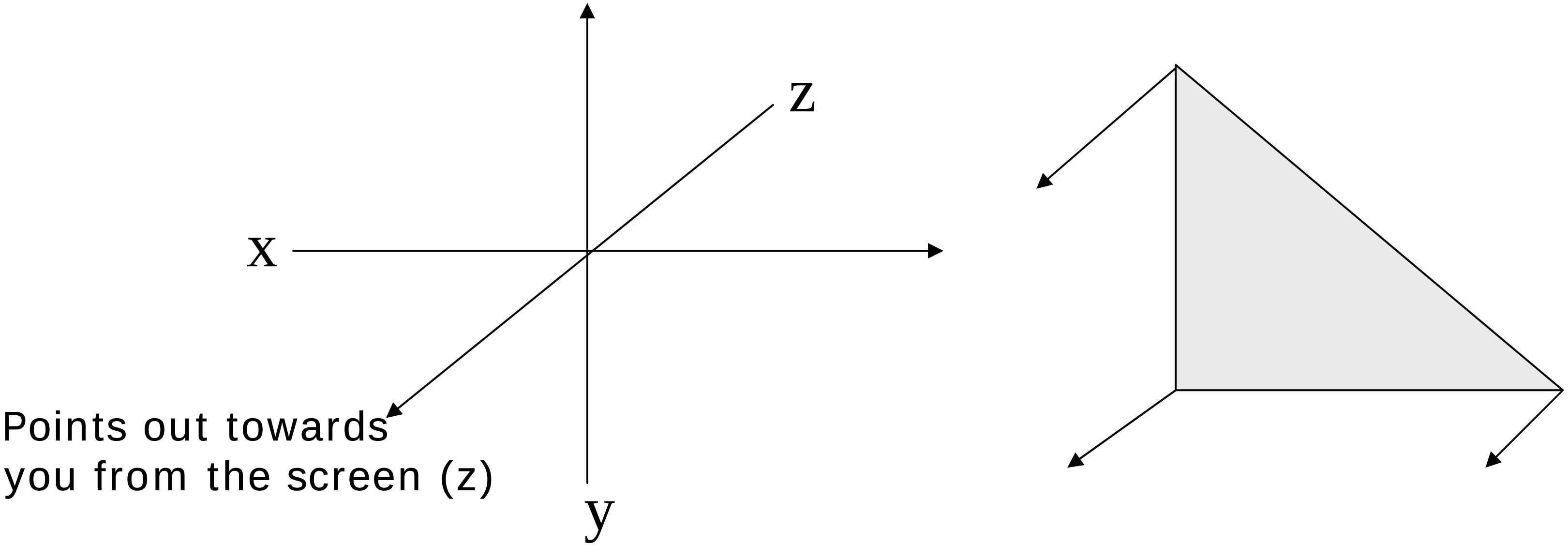
Defining normals of a triangle

Example 2:

8-bit value

```
byte[] normals = { 0, 0, 127,    0, 0, 127,    0, 0, 127};
VertexArray normalsArray = new VertexArray(normals.length /3, 3, 1);
normalsArray.set(0, normals.length/3, normals);
```

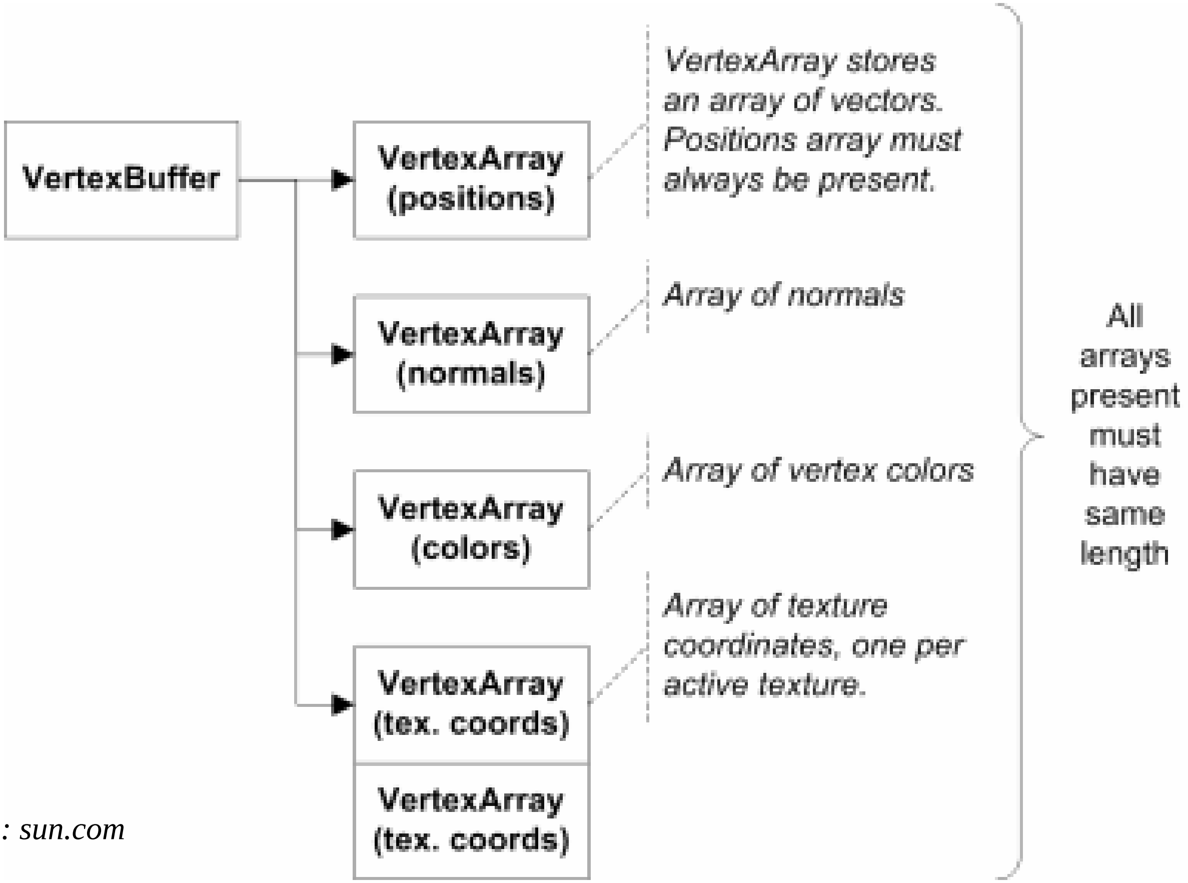
- Normals indicate which side of a triangle gets lighting and effects. (Side of triangle facing you, positive z axis)
- The normal of a surface is always perpendicular (at 90 degrees) to the surface itself.
- Each vertex will have a normal



Points out towards you from the screen (z)

VertexBuffer – Combine vertex info.

- VertexBuffer holds references to VertexArrays that contain the positions, colors, normals, and texture coordinates for a set of vertices.
- The elements of these arrays are called *vertex attributes*.


VertexBuffer
VertexArray (positions)
VertexArray (normals)
VertexArray (colors)
VertexArray (tex. coords)
VertexArray (tex. coords)
VertexArray stores an array of vectors. Positions array must always be present.
Array of normals
Array of vertex colors
Array of texture coordinates, one per active texture.
All arrays present must have same length

Source: sun.com

VertexBuffer – Combine vertex info.

Example:

```
VertexBuffer verbuf = mVertexBuffer = new VertexBuffer();
verbuf.setPositions(vertexArray, 1.0f, null);
verbuf.setNormals(normalsArray);
```

setNormals(VertexArray normals)

Sets the normal vectors for this VertexBuffer.

setPositions(VertexArray positions, **float scale, float[] bias**)

Sets the vertex positions for this VertexBuffer.

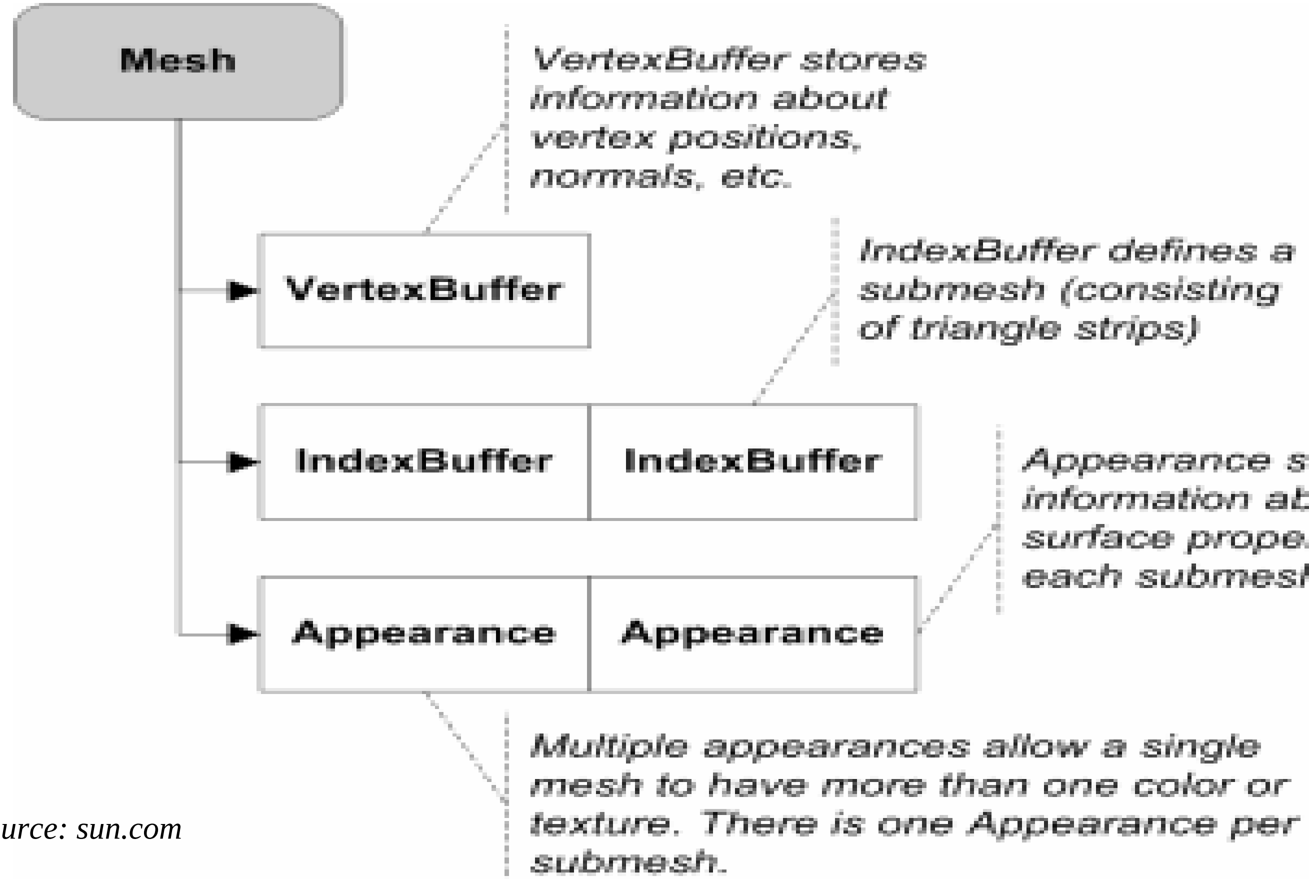
m3g performance optimisation

Other Methods:

VertexArray**getColors**(), int **getDefaultColor**(), VertexArray **getNormals**(), VertexArray **getPositions**(float[] scaleBias), VertexArray **getTexCoords**(int index, float[] scaleBias), int **getVertexCount**(), void **setColors**(VertexArray colors), void **setDefaultColor**(int ARGB) , void **setNormals**(VertexArray normals), void **setPositions**(VertexArray positions, float scale, float[] bias), void **setTexCoords**(int index, VertexArray texCoords, float scale, float[] bias)

Mesh

- Common buffer of vertex data
- An object that can be rendered in M3G is contained in a submesh.
- One or more Appearance for each submesh
- In M3G 1.0, the only submesh available is a **TriangleStripArray** (subclass of IndexBuffer).



VertexBuffer stores information about vertex positions, normals, etc.

IndexBuffer defines a submesh (consisting of triangle strips)

Appearance stores information about the surface properties of each submesh

Multiple appearances allow a single mesh to have more than one color or texture. There is one Appearance per submesh.

Source: sun.com

Mesh

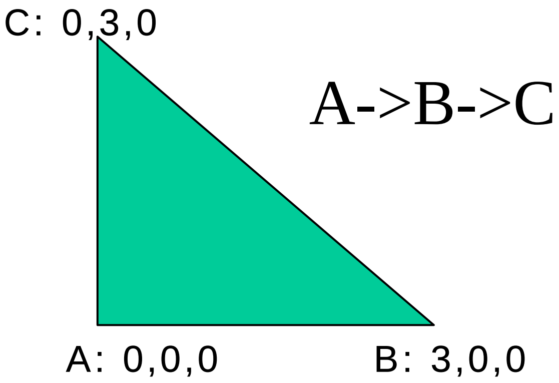
- **Mesh**(VertexBuffer vertices, IndexBuffer[] submeshes, Appearance[] appearances)

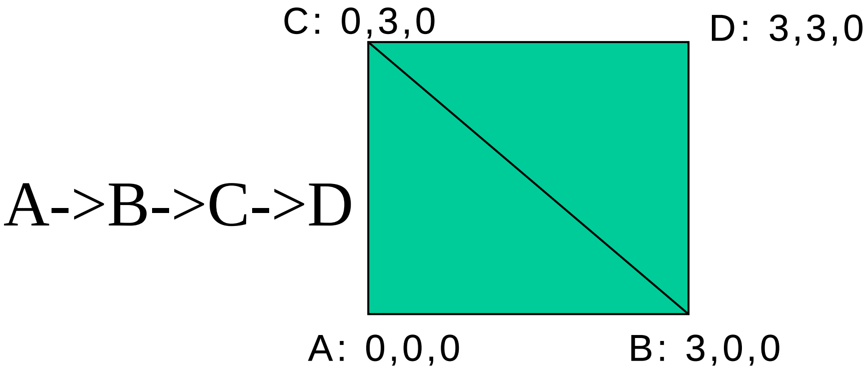
  Constructs a new Mesh with the given VertexBuffer and submeshes.
- **Mesh**(VertexBuffer vertices, IndexBuffer submesh, Appearance appearance)

  Constructs a new Mesh consisting of only one submesh.

Triangle Strips

- An IndexBuffer defines a submesh. TriangleStripArray is a submesh. A triangle strip can represent multiple adjoining triangles. That is a triangle strip is a polygon. TriangleStripArray submesh is a group of polygons.
- The triangle strips are formed by indexing the vertex coordinates and other vertex attributes in an associated VertexBuffer.
- All submeshes in a Mesh share the same VertexBuffer.
- M3G allows for a strip with an arbitrary number of triangles. Which means, a triangle strip can be used to create any arbitrary polygon.
- All triangles in a strip share common side(s) with others.
- *TriangleStripArray* in M3G can use a compact way of specifying vertices for multiple triangles.
  - Three vertices -> one triangle
  - How many vertices are needed to specify 2 adjoining triangles?





Defining a SubMesh with TriangleStripArray

```
int[] stripLength = { 3 };
IndexBuffer mIndexBuffer = new
  TriangleStripArray( 0, stripLength );
```

-a strip array with one triangle (polygon).

- ➢ **TriangleStripArray**(int firstIndex, int[] stripLengths) Constructs a triangle strip array with *implicit* indices.
  - every additional vertex will define a new Triangle.
  - {0,0,0. 3,0,0,   0,3,0,   3,3,0 }   == > Is a square with 2 Triangles. A strip array with 2 Triangles (2 polygons).
- ➢ The TriangeStripArray keeps track of where one strip ends and the next one starts.
- ➢ **Explicit Index**

```
int[] stripLength = { 3, 4, 3 };
IndexBuffer mIndexBuffer = new TriangleStripArray( 0,
 stripLength );
```

  - 3 strips (3 polygons) → (0,1,2), (3,4,5,6), (7,8,9)
  - Constructs a triangle strip array with *explicit* indices.

Giving an Appearance to the Submesh

Groups a set of objects that control how a Submesh will appear when it is rendered.

These objects are called rendering attributes.

```
private Background mBackground = new Background();
private Appearance mAppearance = new Appearance();
private Material mMaterial = new Material();
mMaterial.setColor(Material.DIFFUSE, 0xFF0000);
mMaterial.setColor(Material.SPECULAR, 0xFF0000);
mMaterial.setShininess(100.0f);
mAppearance.setMaterial(mMaterial);
mBackground.setColor(0x00ee88);
```

Material is one of the rendering attribute.

Material

- Controls the color and how light will reflect off the submesh being rendered.

- void **setColor**(int target, int ARGB)

  Sets the given value to the specified color component(s) of this Material. [the alpha component is ignored for all but the diffuse color]

- void **setShininess**(float shininess)

  Sets the shininess of this Material. [0 (dull) to 128 (shiny)]

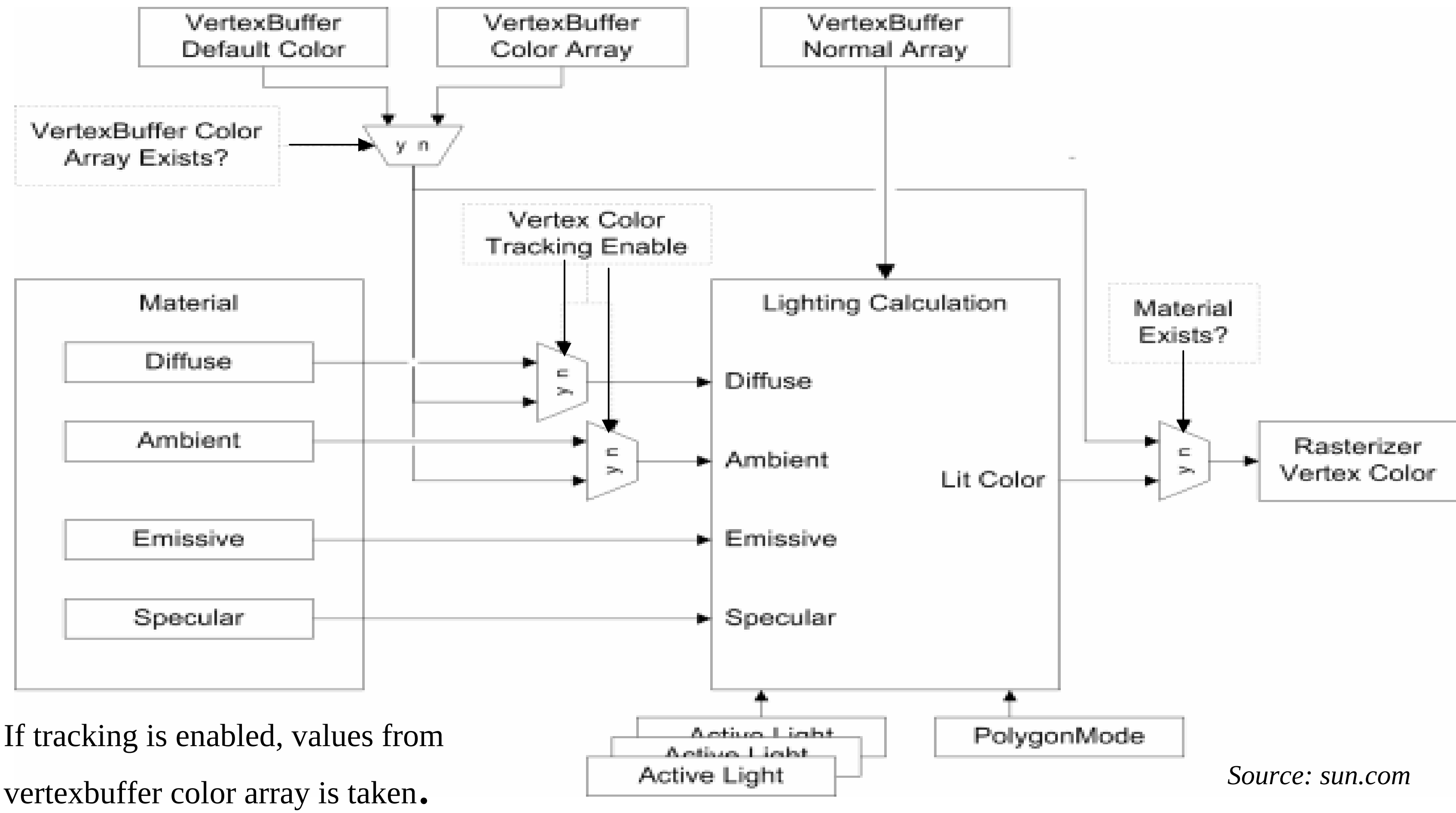
- Targets (Material's color targets):

AMBIENT	The ambient color component, the color of the material that is revealed by ambient (evenly distributed) lighting.
DIFFUSE	The diffuse color component, the color of the material that is revealed by a directional lighting
EMISSIVE	The emission color component, the color of the material that appears to be glowing
SPECULAR	The specular color component, the color displayed in the reflection highlights

- **Other methods:** int **getColor**(int target), float **getShininess**(), boolean **isVertexColorTrackingEnabled**(),
void **setVertexColorTrackingEnable**(boolean enable).

Material

- How the final, *lit* color is obtained for a vertex?
- An Appearance component encapsulating material attributes for lighting computations. Other attributes required for lighting are defined in **Light**, **PolygonMode** and **VertexBuffer**.



If tracking is enabled, values from vertexbuffer color array is taken.

Source: sun.com

Material

- ➢ Default attribute values of Material object

new Material();

- ✶ vertex color tracking : *false* (disabled)
- ✶ ambient color : 0x00333333 (0.2, 0.2, 0.2, 0.0)
- ✶ diffuse color : 0xFFCCCCCC (0.8, 0.8, 0.8, 1.0)
- ✶ emissive color : 0x00000000 (0.0, 0.0, 0.0, 0.0)
- ✶ specular color : 0x00000000 (0.0, 0.0, 0.0, 0.0)
- ✶ shininess : 0.0

Background

- ➢ Backgraound object is an M3G object, that is used to render the background.
- ➢ Note - In retained-mode Background object is a part of World object.
- ➢ Background can be either an Image2D or Color
  - ✶ void setImage(Image2D image);
  - ✶ void setColor(int ARGB)

Camera

- Camera is an m3g class that controls what you see in the rendering
- Camera has a position and rendering.
- Camera attributes: field of view, aspect ratio, and clipping panes. Anything outside the field of view and clipping pane is not computed for rendering.

60 deg field of view (controls how much of the scene you can see)

```
mCamera.setPerspective( 60.0f,
        (float)getWidth()/ (float)getHeight(),
        1.0f, 1000.0f );
```

Near and far clipping panes

Aspect ratio is same as Canvas

Question: If near = far, what will be the View Volume?

Light

- Light (a m3g class) has a position and rendering.
- Light Attributes:- color, different modes (eg spot light vs diffused light), intensity, etc.

```
mLight.setColor(0xffffff);
mLight.setIntensity(1.25f);
```

Brighter than default.

Default mode: directional spot light
Default intensity: 1

Rendering

```
mGraphics3D.clear(mBackground);

mTransform.setIdentity();   //Replaces this transformation with the
    4x4 identity matrix.
mTransform.postTranslate(0.0f, 0.0f, 10.0f);
mGraphics3D.setCamera(mCamera, mTransform);

mGraphics3D.resetLights();
mGraphics3D.addLight(mLight, mTransform);

mAngle += 1.0f;
mTransform.setIdentity();
mTransform.postRotate(mAngle, 0, 1.0f, 0 );
mGraphics3D.render(mVertexBuffer, mIndexBuffer, mAppearance,
    mTransform);
```

Demo

- ➢ Putting it all together
  - ✶ Triangle demo
  - ✶ Square demo

Culling

- Culling is an optimisation technique
  - Culling avoids rendering surfaces that are never shown, and therefore saves on computation required during rendering.
  - Disabling
    - PolygonMode is an m3g class representing a rendering attribute. It is grouped by an Appearance instance and can be used to control culling. [Default – CULL_BACK]

```
PolygonMode tPoly = new PolygonMode();
tPoly.setCulling(PolygonMode.CULL_NONE);
mAppearance.setPolygonMode(tPoly);
```

Demo

Creating a cube with 3 panes

- 3 Strips of a cube (representing 3 sides/panes)
  - x-y, y-z, x,z
- Defining vertexes for the three strips
  - Strip 1 (x-y): (0,0,0),(3,0,0),(0,3,0),(3,3,0)
  - Strip 2 (y-z): (3,0,0),(3,3,0),(3,0,-3),(3,3,-3)
  - Strip 3 (x-z): (0,0,0),(3,0,0),(0,0,-3),(3,0,-3)

```
 short[] vertices = {
            0, 0, 0,  3, 0, 0,  0, 3,  0,  3, 3,  0,
            3, 0, 0,  3, 3, 0,  3, 0, -3,  3, 3, -3,
            0, 0, 0,  3, 0, 0,  0, 0, -3,  3, 0, -3
        };
 VertexArray vertexArray = new VertexArray(vertices.length / 3, 3, 2);
 vertexArray.set(0, vertices.length/3, vertices);
VertexBuffer mVertexBuffer = new VertexBuffer();
mVertexBuffer.setPositions(vertexArray, 1.0f, null);
```

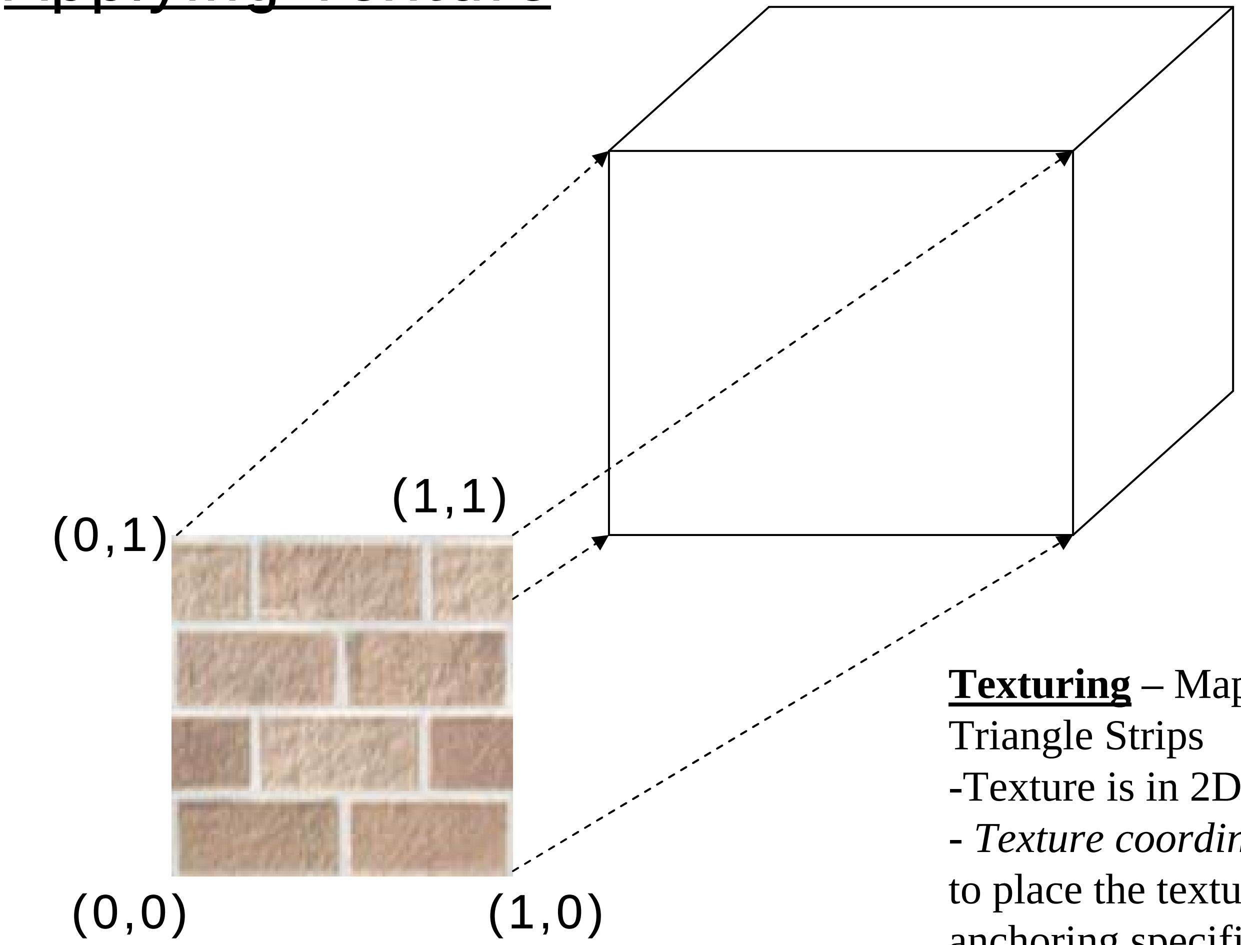
Creating a cube with 3 panes

- Normals corresponding to the vertices

```
byte[] normals = {
        0,      0, 127,    0,     0, 127,     0,   0,  127,    0,     0, 127,
        127,     0,    0,  127,    0,    0,  127,  0,     0,  127,    0,    0,
        0,    -127,   0,     0, -127,   0,      0, -127,  0,      0, -127,    0
    };

VertexArray normalsArray = new VertexArray(normals.length / 3, 3,
    1);
normalsArray.set(0, normals.length/3, normals);
VertexBuffer mVertexBuffer = new VertexBuffer();
mVertexBuffer.setNormals(normalsArray);
```

TextureMapping the Cube Exterior

Applying Texture



Texturing – Mapping an image to Triangle Strips

-Texture is in 2D (2D Image)

- *Texture coordinates* – Tells m3g, how to place the texture on the surface by anchoring specific texture points to the vertices.

TextureMapping the Cube Exterior

Applying Texture

```
short[] texturecords = {
        0,1,    1,1,    0,  0,  1,  0,
        0,1,    0,0,    1,  1,  1,  0,
        0,0,    1,0,    0,  1,  1,  1  };

VertexArray textureArray =
        new VertexArray(texturecords.length / 2, 2, 2);
textureArray.set(0, texturecords.length/2, texturecords);

int[] stripLength = { 4, 4, 4};

VertexBuffer mVertexBuffer = new VertexBuffer();
vertexBuffer.setTexCoords(0, textureArray, 1.0f, null);
```

TextureMapping the Cube Exterior

- Texture coordinates may have either two or three components.
- If the third component is not given, it is implicitly set to zero.
- The components are interpreted in (S, T) or (S, T, R) order, each component being a signed 8-bit or 16-bit integer.
- Texture coordinates have associated with them a uniform scale and a per-component bias, which behave exactly the same way as with vertex positions. Non-uniform scaling is not supported, so as to make texture coordinates behave consistently with vertex positions.

TextureMapping the Cube Exterior

Image for texture

- ✶ Use m3g’s Image2D

  new Image2D(int format, Object image)

  – Format

  » RGB, RGBA, ALPHA, LUMINANCE, LUMINANCE_ALPHA

```
Image mImage;
mImage = Image.createImage(“/texture.png”);
Image2D image2D = new Image2D(Image2D.RGB, mImage);
//adding texture to Appearance
Texture2D texture = new Texture2D( image2D );
mAppearance.setTexture(0, texture);
```

DEMO

Less memory usage: *Use Image2D.Load instead of CreateImage*