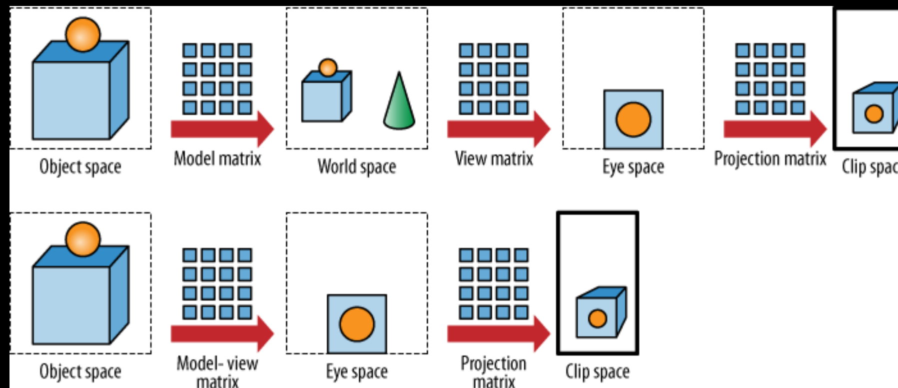
MODEL VIEW AND PERSPECTIVE MATRICES

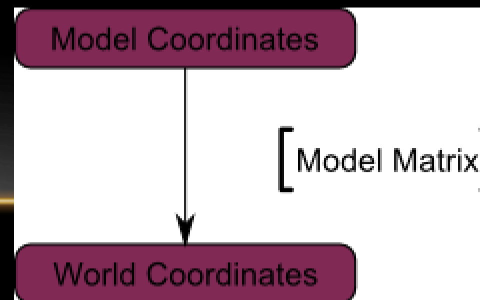


OUTLINE

- The Three Main Matrices
  - Model Matrix
  - View Matrix
  - Projection Matrix
- Instancing
- Multiple Objects

THE MODEL MATRIX

- Positions and orients the object (model) in the world coordinates
- Each object (model) has its own model matrix
  - If that object moves in the scene, then the matrix needs to change
- Assuming objects move, needs to be created for each object at each frame

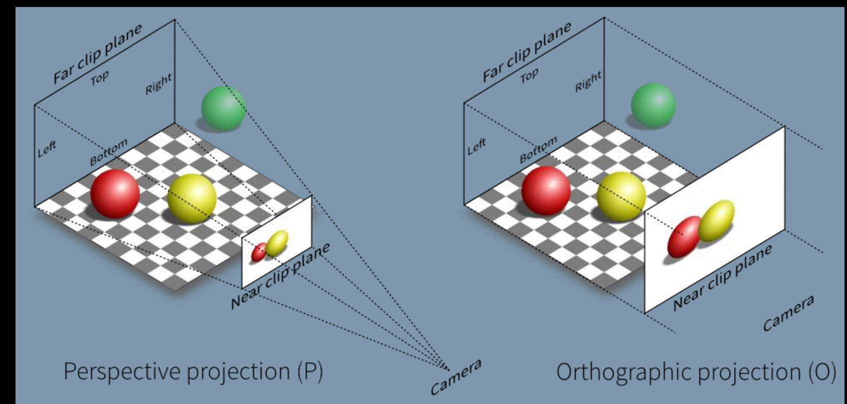


THE VIEW MATRIX

- Moves the "world" – and objects in it – to represent the position of the camera
- Because the OpenGL camera is fixed, if we want to "move" the camera in a particular way, we actually move the world in the opposite way
- Created once per frame, and applies to all objects

THE PROJECTION MATRIX

- Used to define how our 3D world will be viewed on a 2D plane
- Orthographic or perspective projections
- Defines the view volume, which in turn defines the clipping planes
  - Rectangular for orthographic
  - Frustum for perspective
  - Near and far planes
  - Aspect ratio
  - Field of view
- Created only once during the program (unless window is resized)



THE (CODING) STRATEGY

- 1. Build the projection matrix based on how we want to transform 3D to 2D
- 2. Build the view matrix based on camera location
- 3. For each object:
  - A. Build a model matrix based on model's location
  - B. Concatentate (multiply) the model and view matrices into a single model-view matrix
  - Send the model-view matrix and projection matrices to the shader(s) as uniform variables

LET'S DRAW A CUBE!!!

PERSPECTIVE PROJECTION MATRIX UTILITY METHOD – CREATE ONCE, CALL FROM INIT()

```
private Matrix3D perspective(float fovy, float aspect, float n, float f)
{
        float q = 1.0f / ((float) Math.tan(Math.toRadians(0.5f * fovy)));
        float A = q / aspect;
        float B = (n + f) / (n - f);
        float C = (2.0f * n * f) / (n - f);
        Matrix3D r = new Matrix3D();
        r.setElementAt(0,0,A);
        r.setElementAt(1,1,q);
        r.setElementAt(2,2,B);
        r.setElementAt(3,2,-1.0f);
        r.setElementAt(2,3,C);
        r.setElementAt(3,3,0.0f);
        return r;
}
```

$$q = \frac{1}{\tan\left(\frac{fieldOfView}{2}\right)}$$

$$A = \frac{q}{aspectRatio}$$

$$B = \frac{Z_{near} + Z_{far}}{Z_{near} - Z_{far}}$$

$$C = \frac{2 * (Z_{near} * Z_{far})}{Z_{near} - Z_{far}}$$

$$\begin{bmatrix} A & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & B & C \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

CAMERA AND OBJECT LOCATIONS

- Let's say we want to "move" our camera back 8 units on the z axis
- And we want to move our object lower on the y axis so that we can see the top surface
- In init():

```
cameraX = 0.0f; cameraY = 0.0f; cameraZ = 8.0f;
cubeLocX = 0.0f; cubeLocY = -2.0f; cubeLocZ = 0.0f;
```

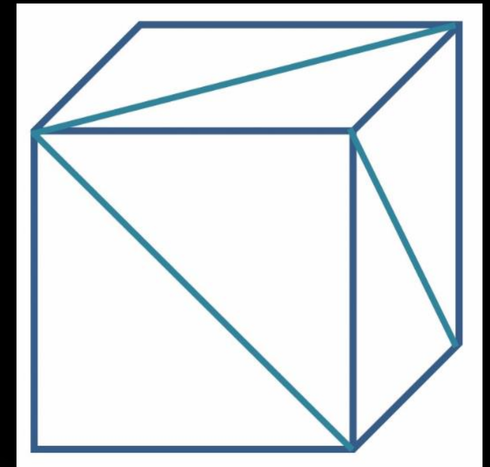
- (camera and cubeLoc variables have been defined as instance variables of type float)

BUILD THE MODEL-VIEW MATRIX (IN DISPLAY())

```
// Opposite of where we want camera
Matrix3D vMat = new Matrix3D();
vMat.translate(-cameraX, -cameraY, -cameraZ);
// Where we want our cube
Matrix3D mMat = new Matrix3D();
mMat.translate(cubeLocX, cubeLocY, cubeLocZ);
// Combine the model and view into model-view
Matrix3D mvMat = new Matrix3D();
mvMat.concatenate(vMat);
mvMat.concatenate(mMat);
// Note: translate(x,y,z) is a utility method from
// graphicslib3D operating on the Matrix3D data type,
// and so is concatenate
```

HOW DO WE DEFINE OUR OBJECT? (1/2)

```
private void setupVertices()
{
		GL4 gl = (GL4) GLContext.getCurrentGL();
		float[] vertex_positions =
		{		-1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f, 1.0f, -1.0f, -1.0f,
				1.0f, -1.0f, -1.0f, 1.0f,  1.0f, -1.0f, -1.0f,  1.0f, -1.0f,
				1.0f, -1.0f, -1.0f, 1.0f, -1.0f,  1.0f, 1.0f,  1.0f, -1.0f,
				1.0f, -1.0f,  1.0f, 1.0f,  1.0f,  1.0f, 1.0f,  1.0f, -1.0f,
				1.0f, -1.0f,  1.0f, -1.0f, -1.0f,  1.0f, 1.0f,  1.0f,  1.0f,
				-1.0f, -1.0f,  1.0f, -1.0f,  1.0f,  1.0f, 1.0f,  1.0f,  1.0f,
				-1.0f, -1.0f,  1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f,  1.0f,
				-1.0f, -1.0f, -1.0f, -1.0f,  1.0f, -1.0f, -1.0f,  1.0f,  1.0f,
				-1.0f, -1.0f,  1.0f,  1.0f, -1.0f,  1.0f,  1.0f, -1.0f, -1.0f,
				1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f, -1.0f,  1.0f,
				-1.0f,  1.0f, -1.0f, 1.0f,  1.0f, -1.0f, 1.0f,  1.0f,  1.0f,
				1.0f,  1.0f,  1.0f, -1.0f,  1.0f,  1.0f, -1.0f,  1.0f, -1.0f
		};
…
```



HOW DO WE DEFINE OUR OBJECT? (2/2)

…

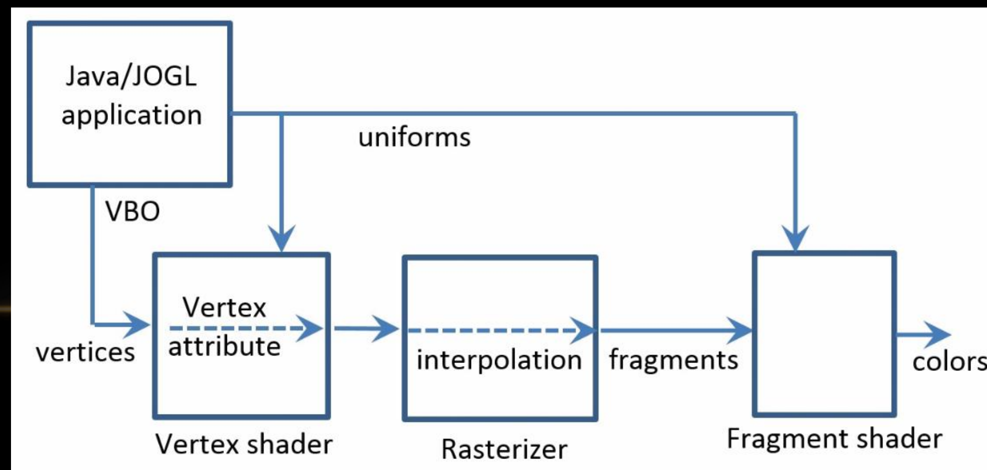
```
gl.glGenVertexArrays(vao.length, vao, 0);
gl.glBindVertexArray(vao[0]);
gl.glGenBuffers(vbo.length, vbo, 0);

gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
FloatBuffer vertBuf =
    Buffers.newDirectFloatBuffer(vertex_positions);
gl.glBufferData(GL_ARRAY_BUFFER, vertBuf.limit()*4,
    vertBuf, GL_STATIC_DRAW);
}
```

WHAT HAPPENS IN THE SHADERS? (1/2) VERTEX SHADER

```
#version 430

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
        gl_Position = proj_matrix * mv_matrix *
                      vec4(position,1.0);
}
```

WHAT HAPPENS IN THE SHADERS? (1/2) FRAGMENT SHADER

```
#version 430

out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
        color = vec4(1.0, 0.0, 0.0, 1.0);
}
```



GETTING THE MATRICES INTO THE SHADERS (IN DISPLAY())

```
int mv_loc = gl.glGetUniformLocation(rendering_program,
                    "mv_matrix");
int proj_loc = gl.glGetUniformLocation(rendering_program,
                                        "proj_matrix");

gl.glUniformMatrix4fv(mv_loc, 1, false,
mvMat.getFloatValues(), 0);
gl.glUniformMatrix4fv(proj_loc, 1, false,
pMat.getFloatValues(), 0);

gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
gl.glEnableVertexAttribArray(0);
```

AND A FEW NEW THINGS FOR OPENGL SETTINGS (STILL IN DISPLAY())

```
// Fill the depth buffer with default values (so objects are
// rendered correctly on each iteration
gl.glClear(GL_DEPTH_BUFFER_BIT);

…

// Enable depth testing (this is a 3D model…)
gl.glEnable(GL_DEPTH_TEST);
// Use the less than or equal depth test
gl.glDepthFunc(GL_LEQUAL);

// Yay! Finally draw something!!!
gl.glDrawArrays(GL_TRIANGLES, 0, 36);
```

OMG!!! WE HAVE A RED CUBE!!!

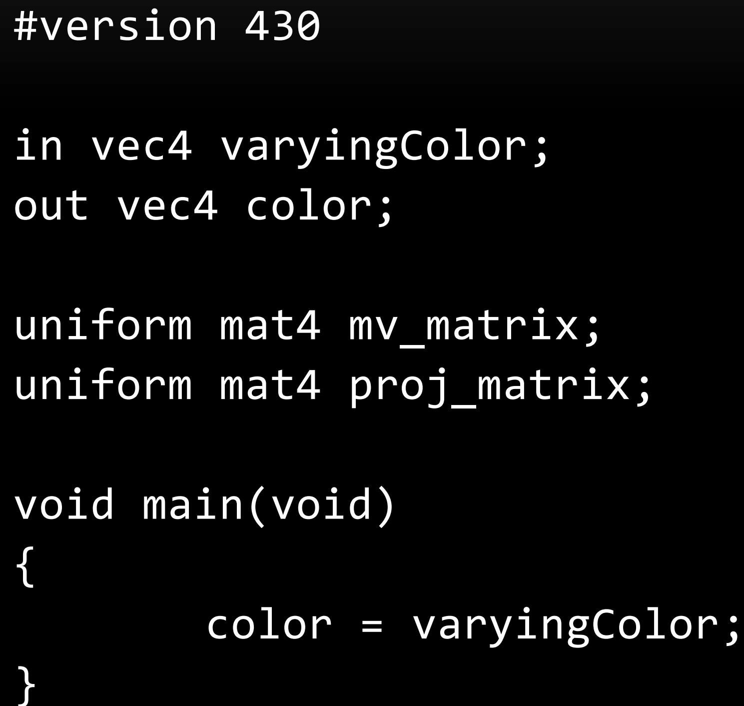
BUT… A RED CUBE IS BORING … WE WANT MORE COLORS

- Let’s assign colors based on position, instead of a flat color
- Can do this by just modifying the shaders

VERTEX SHADER

```
#version 430

layout (location=0) in vec3 position;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec4 varyingColor;

void main(void)
{
  gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
  varyingColor = vec4(position,1.0)*0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}
```

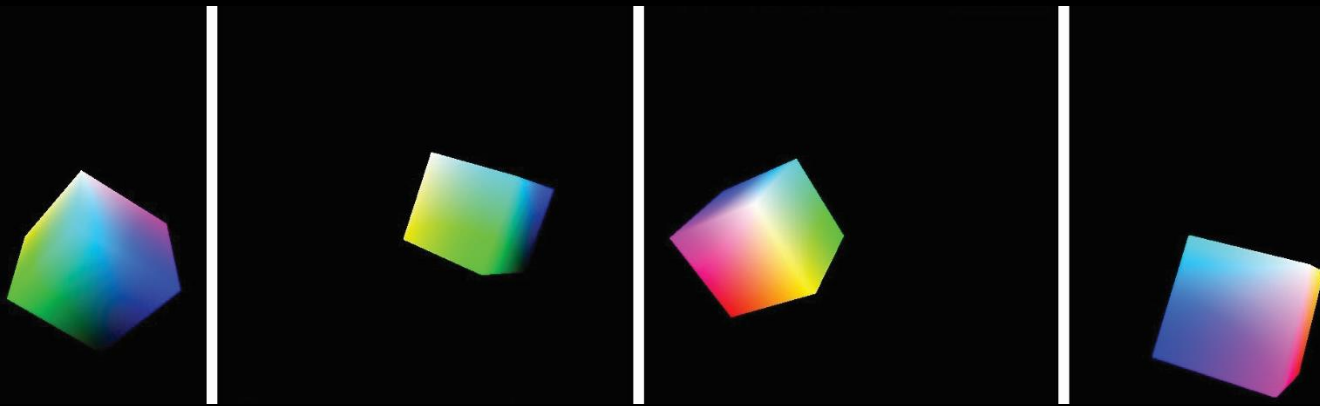
FRAGMENT SHADER



```
#version 430

in vec4 varyingColor;
out vec4 color;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

void main(void)
{
	color = varyingColor;
}
```

UM … STILL BORING … CAN WE MAKE IT MOVE?

- Yes.
- Well. Movement doesn't show well on slides…



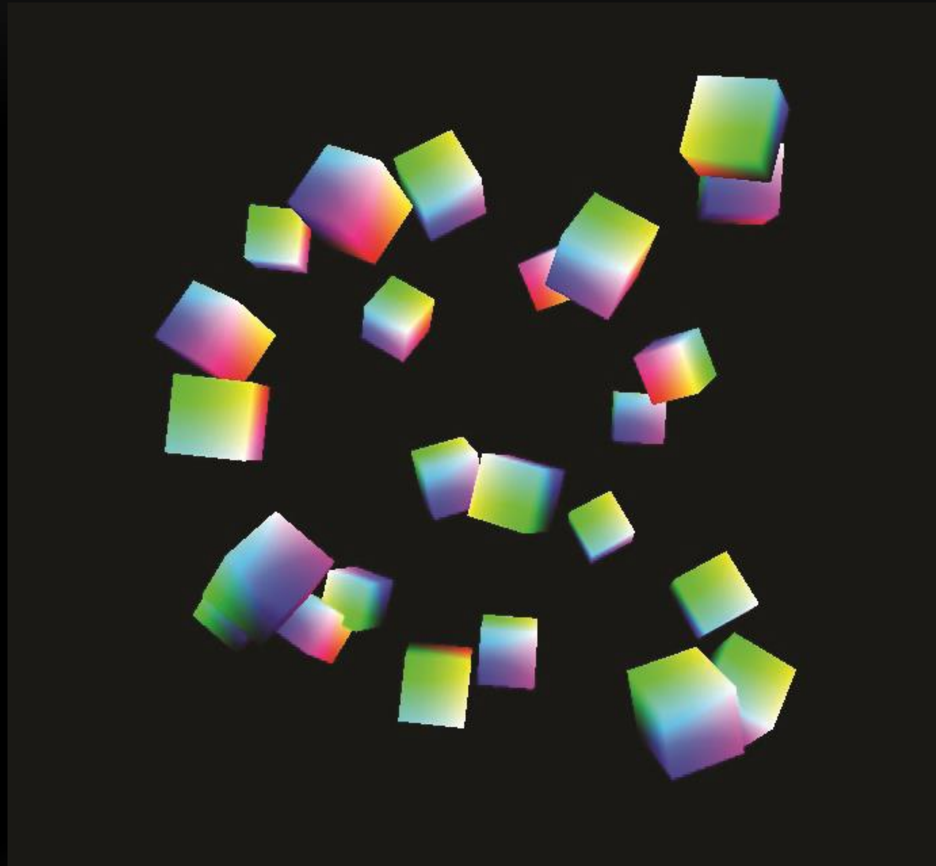
WE DO THIS ON THE JAVA/JOGL SIDE

```
// In display – clear depth and background at each iteration
gl.glClear(GL_DEPTH_BUFFER_BIT);
float bkg[] = { 0.0f, 0.0f, 0.0f, 1.0f };
FloatBuffer bkgBuffer = Buffers.newDirectFloatBuffer(bkg);
gl.glClearBufferfv(GL_COLOR, 0, bkgBuffer);
// Create a model matrix that translates and rotates based
// on the system time
Matrix3D mMat = new Matrix3D();
double x = (double) (System.currentTimeMillis())/10000.0;
mMat.translate(Math.sin(2*x)*2.0, Math.sin(3*x)*2.0,
Math.sin(4*x)*2.0);
mMat.rotate(1000*x, 1000*x, 1000*x);
```

CREATE AN FPSANIMATOR IN THE CONSTRUCTOR AS WE DID WITH THE POINT

```
FPSAnimator animator = new FPSAnimator(myCanvas, 30);
animator.start();
```

COOL! OK … WELL, STILL BORING … CAN WE HAVE MULTIPLE CUBES?



INSTANCING

- Instancing allows us to make multiple copies with different transformations of the same object
  - Instead of using glDrawArrays,() we can use glDrawArraysInstanced()
  - eg. glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 24);
    - This gets us 24 of our objects – and make model changes in shader
  - If we want each to be positioned and move independently, we need to separate the model and view matrices
    - View remains the same, but the model matrix changes
  - We can make changes to the model matrix in either the Java/JOGL code or the vertex shader

JAVA/JOGL CODE APPROACH

```
double timeFactor = (double) (System.currentTimeMillis()%3600000)/10000.0;

for (int i=0; i<24; i++)
{
    double x = i + timeFactor;

    Matrix3D mMat = new Matrix3D();

    mMat.translate(Math.sin(2*x)*6.0,
    Math.sin(3*x)*6.0,
    Math.sin(4*x)*6.0);
    mMat.rotate(1000*x, 1000*x, 1000*x);

    Matrix3D mvMat = new Matrix3D();
    mvMat.concatenate(vMat);
    mvMat.concatenate(mMat);

    gl.glUniformMatrix4fv(mv_loc, 1, false, mvMat.getFloatValues(), 0);

    gl.glBindBuffer(GL_ARRAY_BUFFER, vbo[0]);
    gl.glVertexAttribPointer(0, 3, GL_FLOAT, false, 0, 0);
    gl.glEnableVertexAttribArray(0);

    gl.glEnable(GL_DEPTH_TEST);
    gl.glDepthFunc(GL_LEQUAL);

    gl.glDrawArrays(GL_TRIANGLES, 0, 36);
}
```

VERTEX SHADER APPROACH

```
#version 430

layout (location=0) in vec3 position;

uniform mat4 m_matrix;
uniform mat4 v_matrix;
uniform mat4 proj_matrix;
uniform float tf;

out vec4 varyingColor;

mat4 buildRotateX(float rad);
mat4 buildRotateY(float rad);
mat4 buildRotateZ(float rad);
mat4 buildTranslate(float x, float y, float z);

void main(void)
{
   float a = sin(2.0 * i) * 8.0;// when 24 instances
   float b = cos(3.0 * i) * 8.0;
   float c = sin(4.0 * i) * 8.0;
   // These functions are defined from Ch. 3
   mat4 localRotX = buildRotateX(1000*i);
   mat4 localRotY = buildRotateY(1000*i);
   mat4 localRotZ = buildRotateZ(1000*i);
   mat4 localTrans = buildTranslate(a,b,c);

   mat4 newM_matrix = m_matrix * localTrans * localRotX * localRotY * localRotZ;
   mat4 mv_matrix = v_matrix * newM_matrix;
   gl_Position = proj_matrix * mv_matrix * vec4(position,1.0);
   varyingColor = vec4(position,1.0)*0.5 + vec4(0.5, 0.5, 0.5, 0.5);
}
```

TWO OF THE FUNCTIONS IN THE VERTEX SHADER – NOTE THE COLUMN ORDER

```
mat4 buildTranslate(float x, float y, float z)
{
   mat4 trans = mat4(1.0, 0.0, 0.0, 0.0,
                     0.0, 1.0, 0.0, 0.0,
                     0.0, 0.0, 1.0, 0.0,
                     x, y, z, 1.0 );
   return trans;
}

mat4 buildRotateX(float deg)
{
   float rad = radians(deg);
   mat4 xrot = mat4(1.0, 0.0, 0.0, 0.0,
                    0.0, cos(rad), -sin(rad), 0.0,
                    0.0, sin(rad), cos(rad), 0.0,
                    0.0, 0.0, 0.0, 1.0 );
   return xrot;
}
```

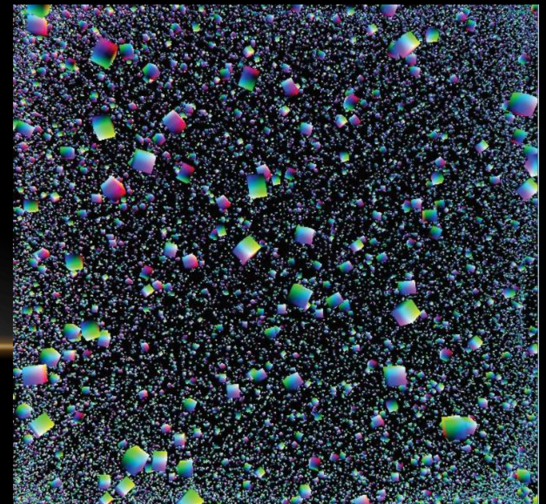
WOW – 24 CUBES! BUT WHAT IF I WANT 100,000 OF THEM?

- In Java/JOGL:

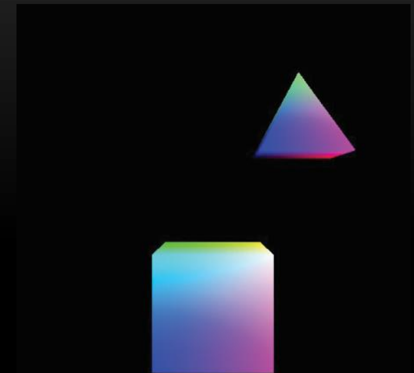
```
gl.glDrawArraysInstanced(GL_TRIANGLES, 0, 36, 100000);
```

- In the vertex shader:

```
//when 100000 instances
float a = sin(203.0 * i/4000.0) * 403.0;
float b = cos(301.0 * i/2001.0) * 401.0;
float c = sin(400.0 * i/3003.0) * 405.0;
```



THAT'S NICE, BUT WHAT IF I WANT OBJECTS THAT ARE DIFFERENT?



- Need to use a separate buffer for each object
- Each object will have its own model matrix
  - But the same view matrix
- Need to generate a different model-view matrix for each
- If the objects are built from the same primitives, can use the same shader program for each
  - Implies that you will need different shader programs for different objects if they are built from different primitives (eg. lines instead of triangles)
- Changes are minimal – pp. 86-87 in text highlight changes

SUMMARY

- The Three Main Matrices
  - Model Matrix
  - View Matrix
  - Projection Matrix
- Instancing
- Multiple Objects