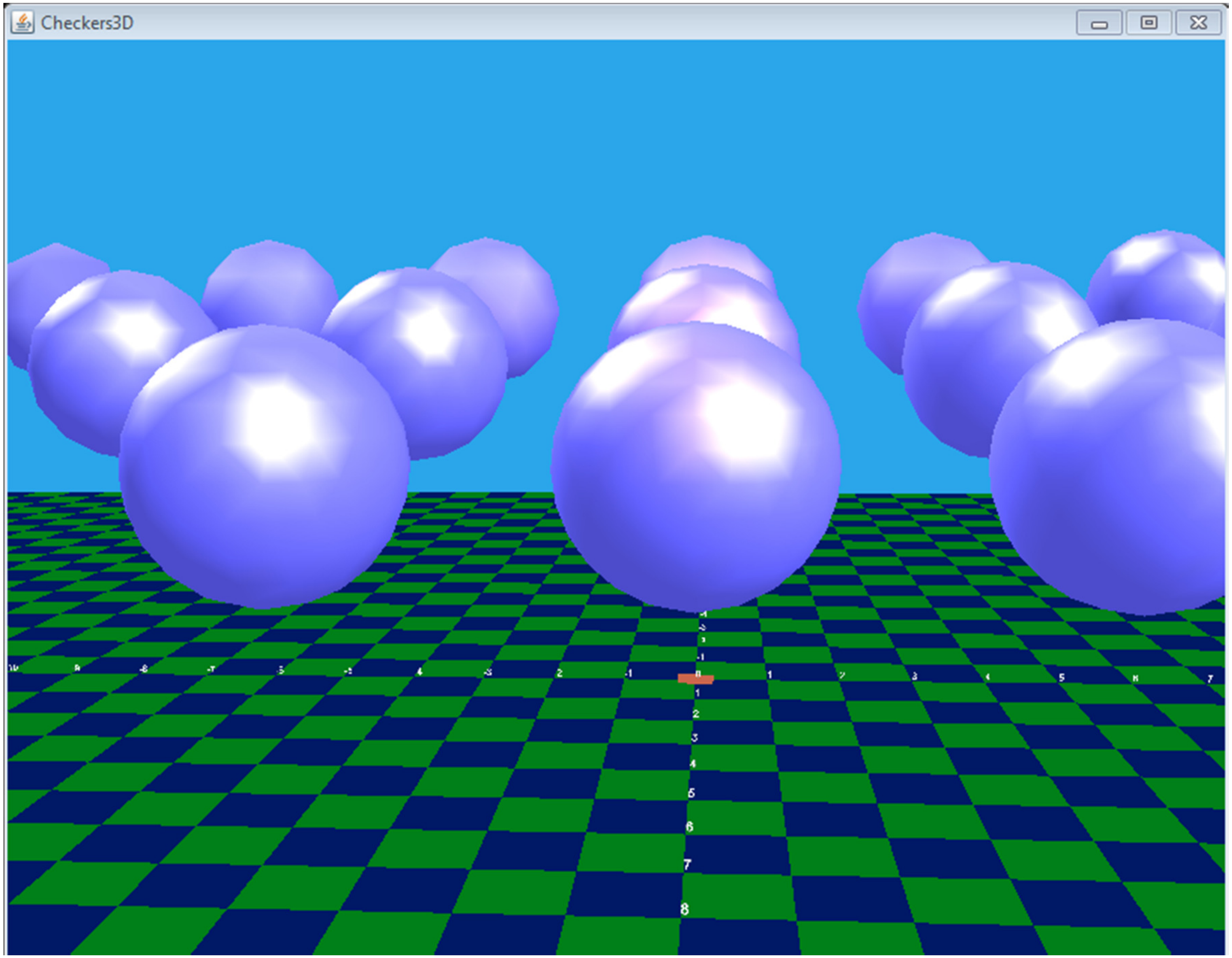
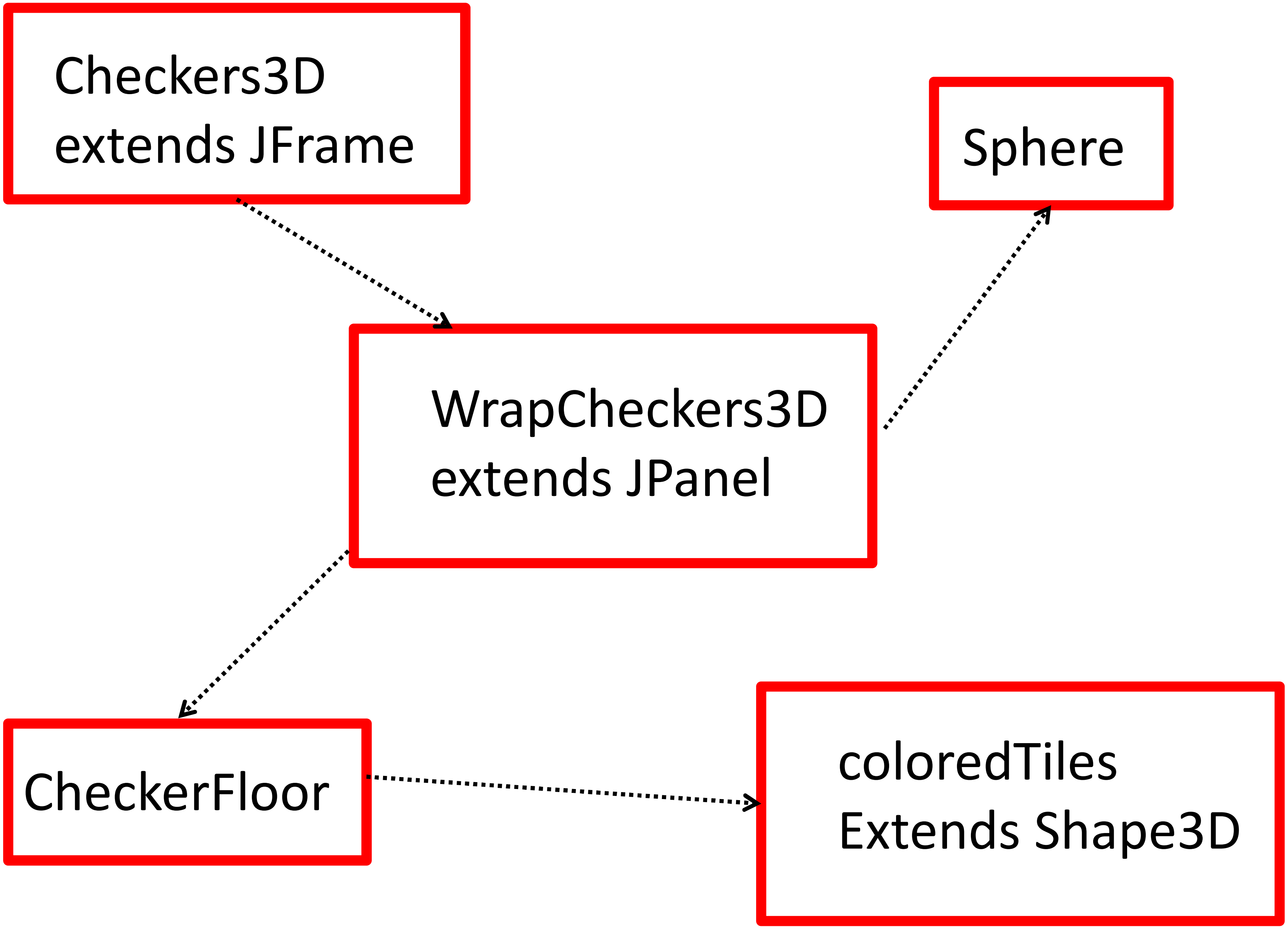
CS324e - Elements of Graphics and Visualization

Checkerboard World

Larger Example - From KGPJ



Classes (Not All)



Checkers3D
extends JFrame

Sphere

WrapCheckers3D
extends JPanel

CheckerFloor

coloredTiles
Extends Shape3D

Checkers3D

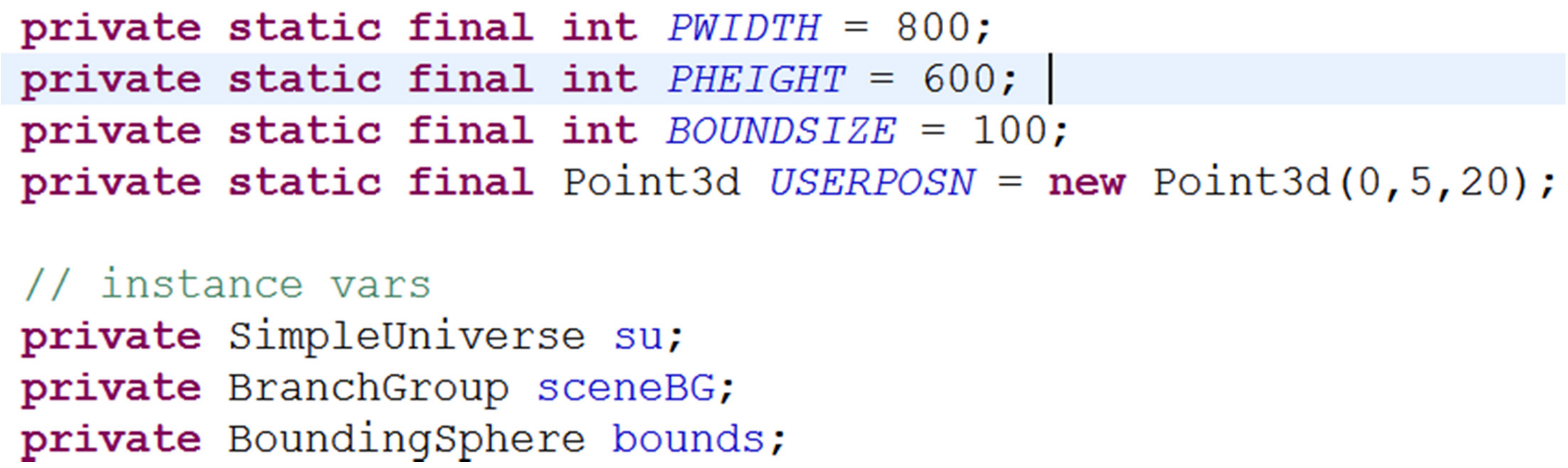
- Extends JFrame
- Similar to frame from HelloUniverse
- Contains the panel that contains the canvas3D
- Could add other GUI components
  - controls or menu items to affect the 3d scene

WrapCheckers3D

- extends JPanel
- Contains the Canvas3D
- Canvas3D a GUI component
- Canvas3D show up on top of other swing components if you try and mix them
  - doesn't play well
  - keep separate
  - don't try to put buttons or scroll bars in canvas
- No animation loop
  - Canvas3D and Scene graph self monitor and if something changes redraw automatically

WrapCheckers3D

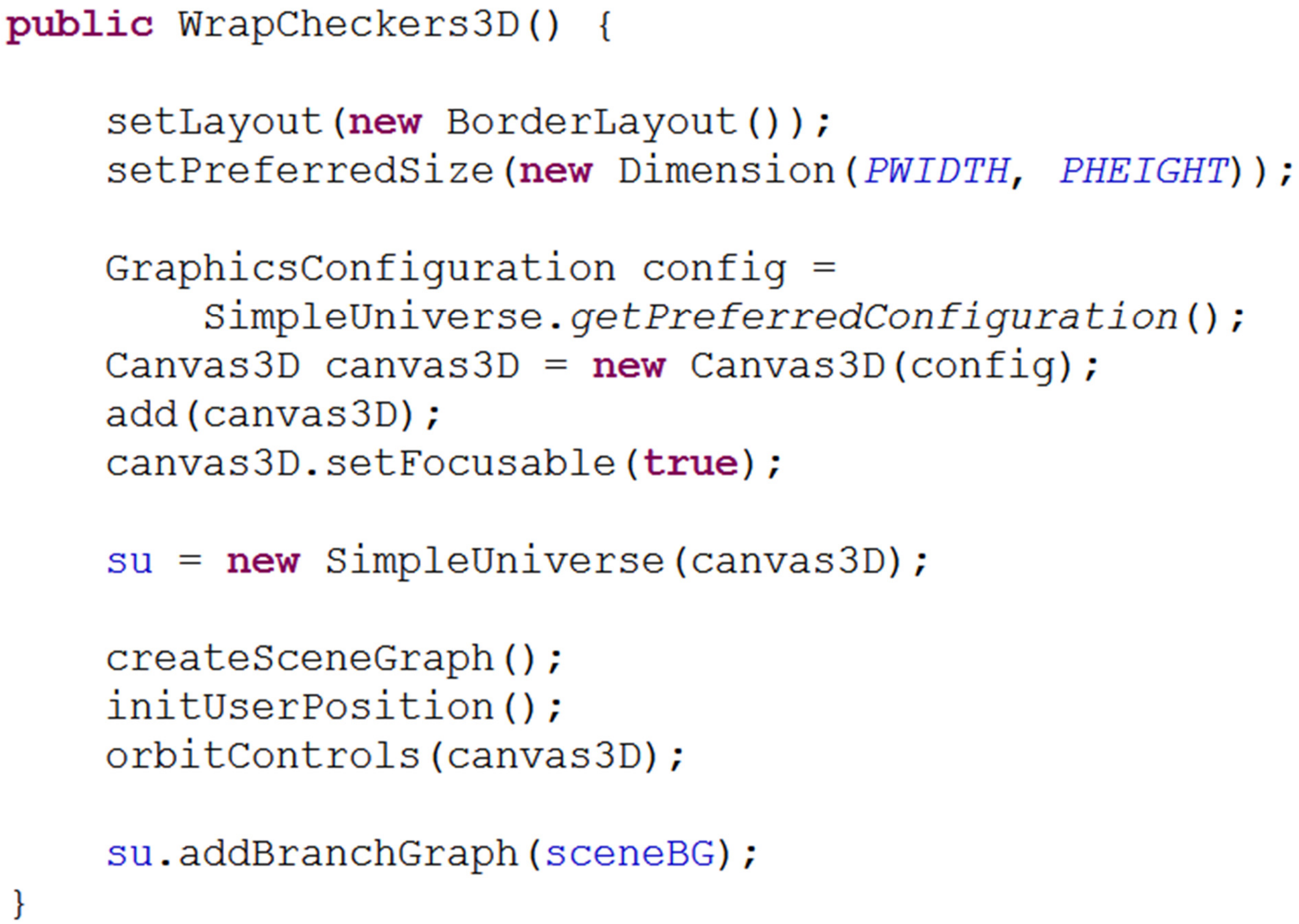
- Most of the code to set up the 3D world
- Instance variables and class constants



```
private static final int PWIDTH = 800;
private static final int PHEIGHT = 600;
private static final int BOUNDSIZE = 100;
private static final Point3d USERPOSN = new Point3d(0,5,20);

// instance vars
private SimpleUniverse su;
private BranchGroup sceneBG;
private BoundingSphere bounds;
```

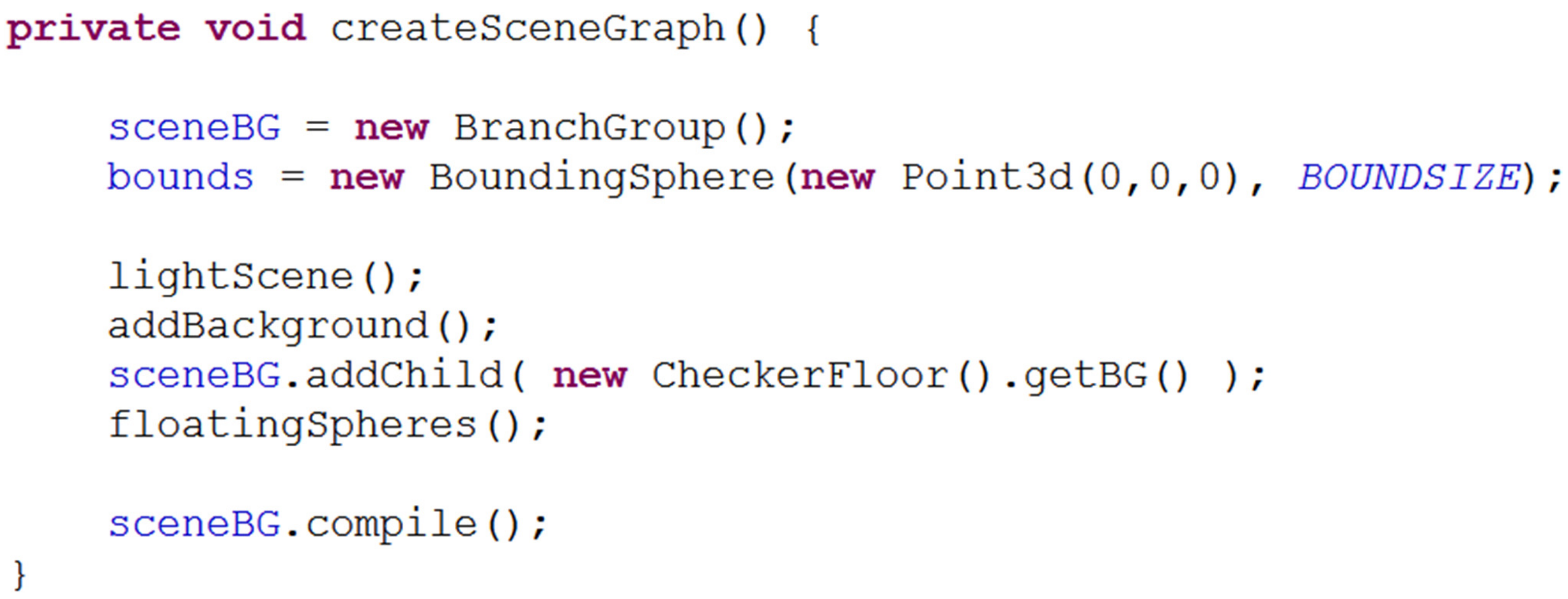
- Where is USERPOSN located?

Creating the World



```java
public WrapCheckers3D() {

    setLayout(new BorderLayout());
    setPreferredSize(new Dimension(PWIDTH, PHEIGHT));

    GraphicsConfiguration config =
        SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas3D = new Canvas3D(config);
    add(canvas3D);
    canvas3D.setFocusable(true);

    su = new SimpleUniverse(canvas3D);

    createSceneGraph();
    initUserPosition();
    orbitControls(canvas3D);

    su.addBranchGraph(sceneBG);
}
```

Creating the Scene Graph



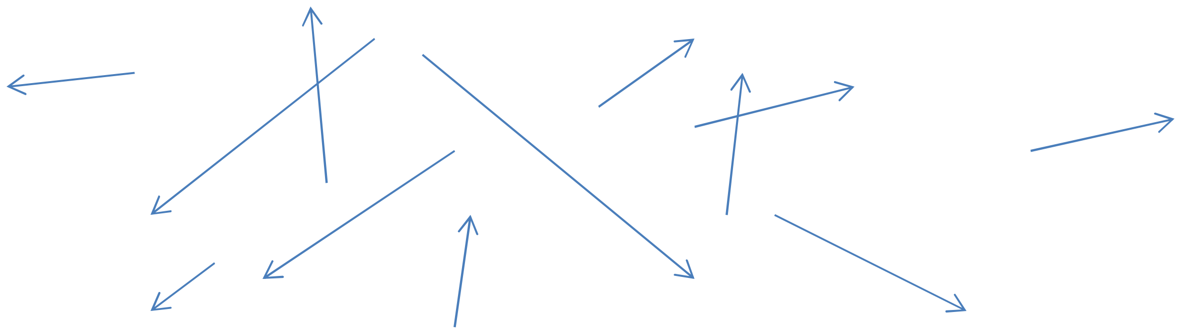
```java
private void createSceneGraph() {

    sceneBG = new BranchGroup();
    bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

    lightScene();
    addBackground();
    sceneBG.addChild( new CheckerFloor().getBG() );
    floatingSpheres();

    sceneBG.compile();
}
```

Lighting the Scene

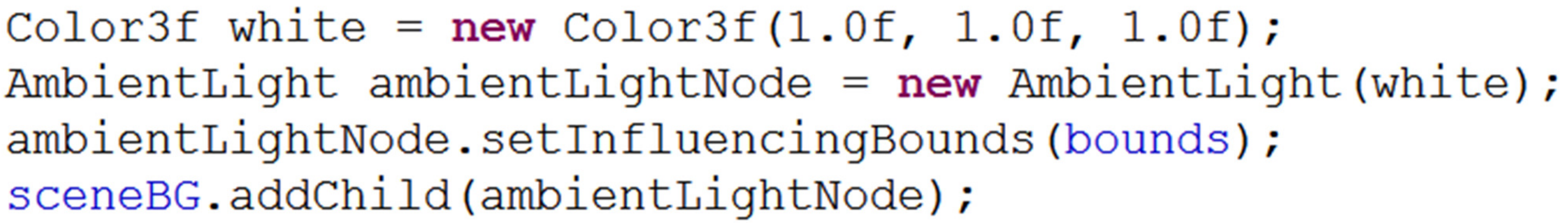
- four kinds of lights can be placed in a Java3D world
  - ambient light
  - directional lights
  - point lights
  - spot lights
- Scene can have multiple lights
- Lights have color, position (possibly), direction (possibly), attenuation (possibly) attributes

Lights

- Ambient lights
  - Uniform in all directions and locations

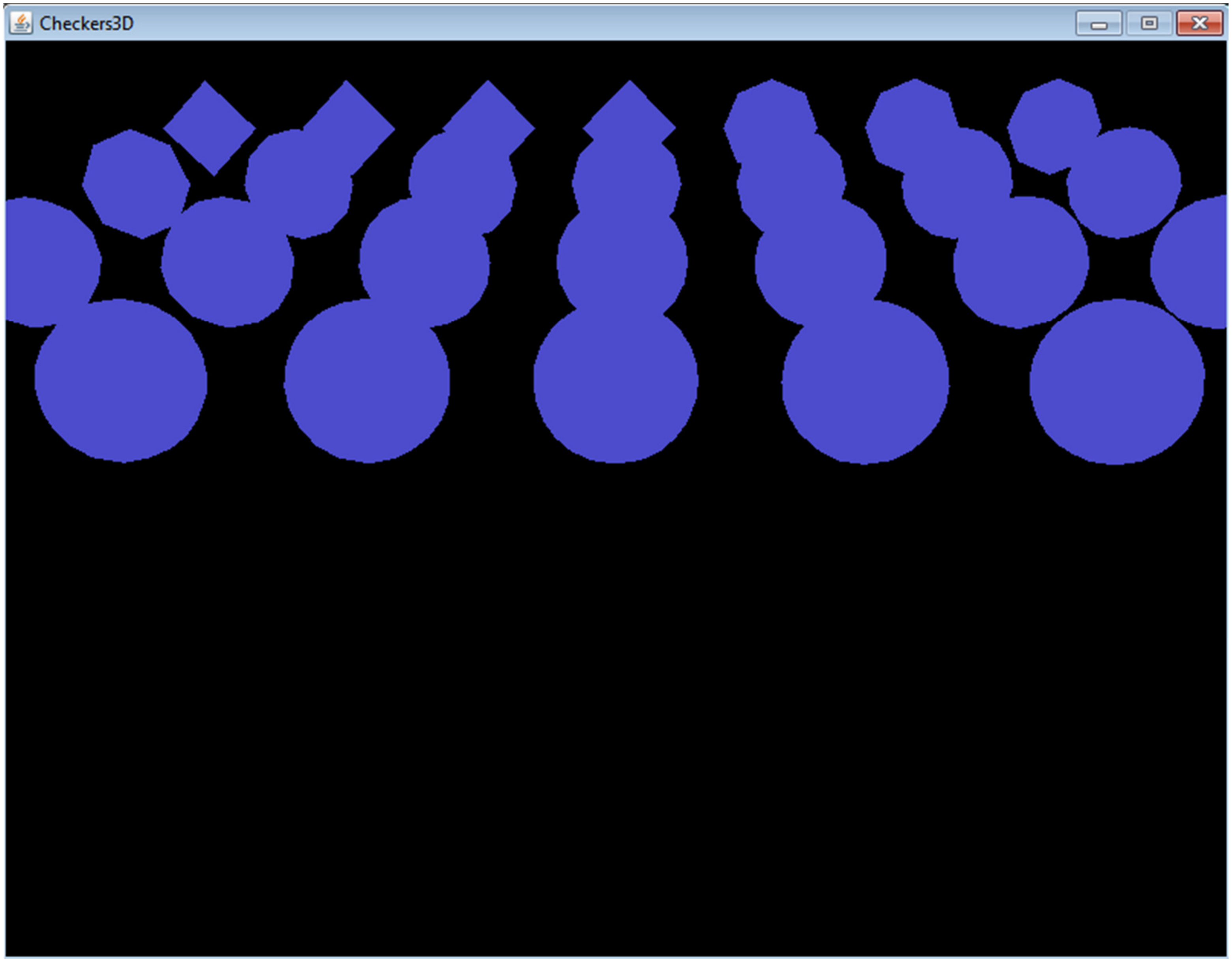


  - create AmbientLight object, give it a color, and add as a node to scene graph
  - Color3f each channel (red, green, blue defined with value between 0 and 1



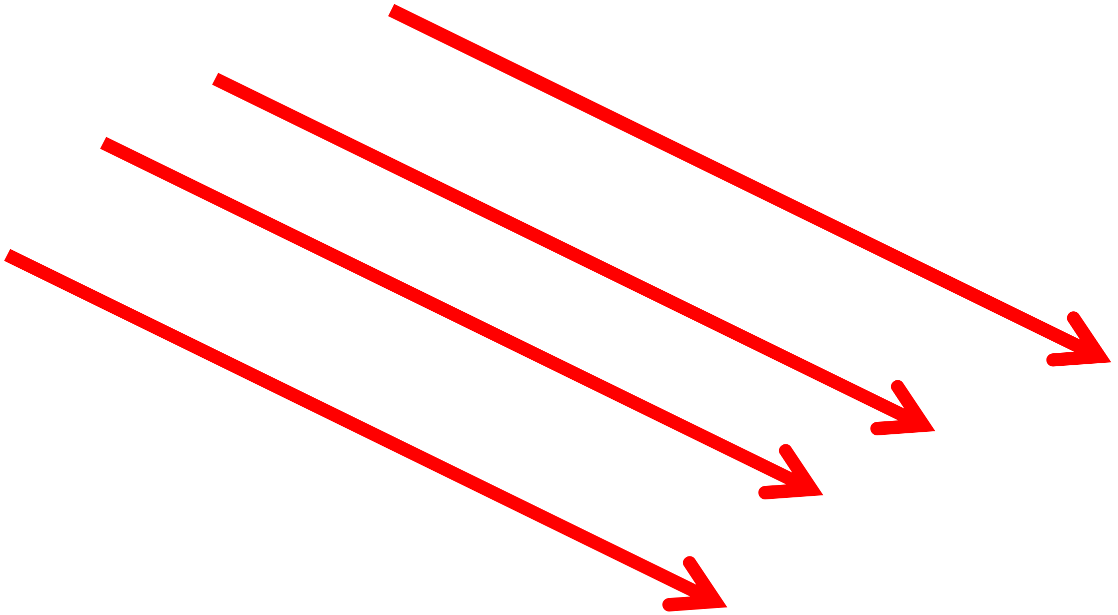
```
Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
AmbientLight ambientLightNode = new AmbientLight(white);
ambientLightNode.setInfluencingBounds(bounds);
sceneBG.addChild(ambientLightNode);
```

Spheres and Ambient Light



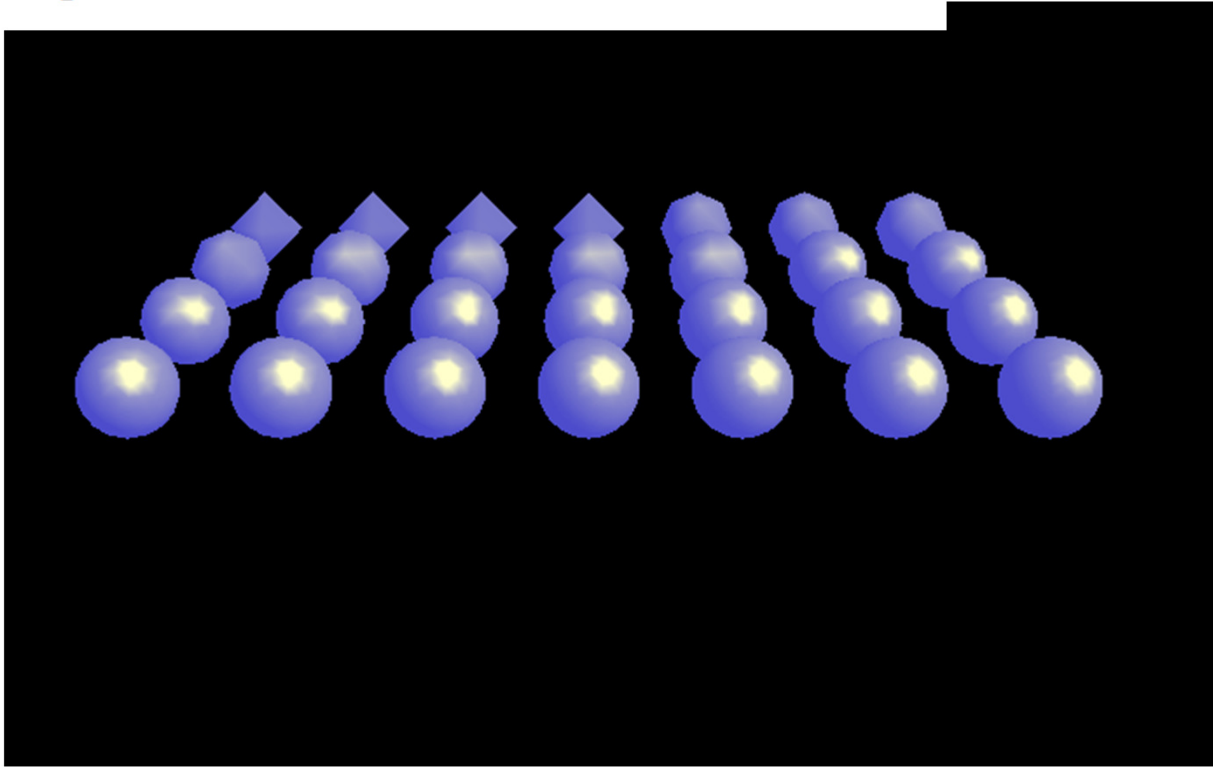
Directional Lights

- fixed direction
- no specific location (think of it as being at an infinite distance away from scene)
- light rays are parallel



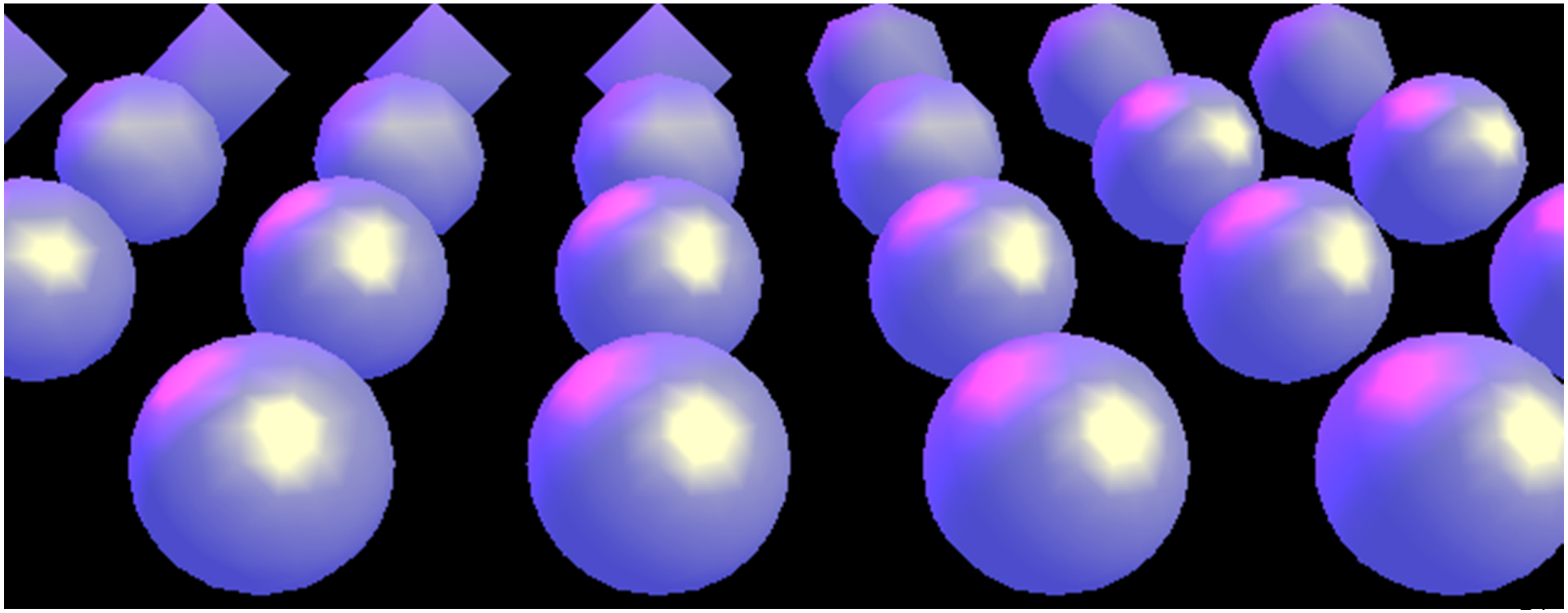
Directional Light

```
// pointing left, down, into scene
Vector3f light1Direction
    = new Vector3f(-1.0f, -1.0f, -1.0f);

Color3f yellow = new Color3f(1, 1, 0);
DirectionalLight light1 =
    new DirectionalLight(yellow, light1Direction);

light1.setInfluencingBounds(bounds);
sceneBG.addChild(light1);
```



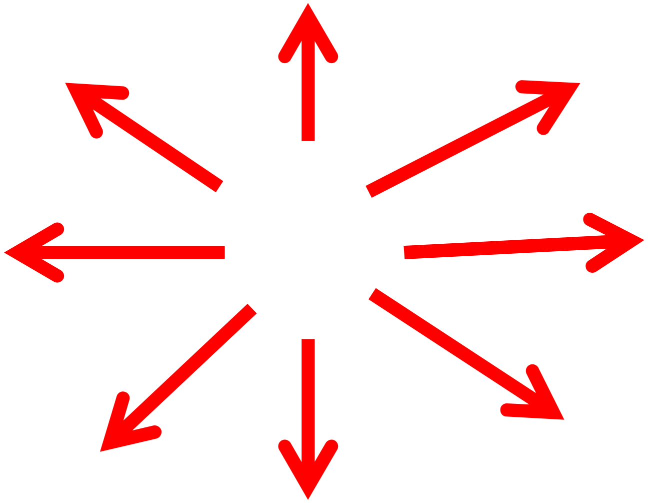
Directional Light

```
// point right, down, out of scene
Vector3f light2Direction
    = new Vector3f(1.0f, -1.0f, 1.0f);

Color3f magenta = new Color3f(1, 0, 1);
DirectionalLight light2 =
    new DirectionalLight(magenta, light2Direction);

light2.setInfluencingBounds(bounds);
sceneBG.addChild(light2);
```



Point Light

- Has a location in the scene
- emits in all directions

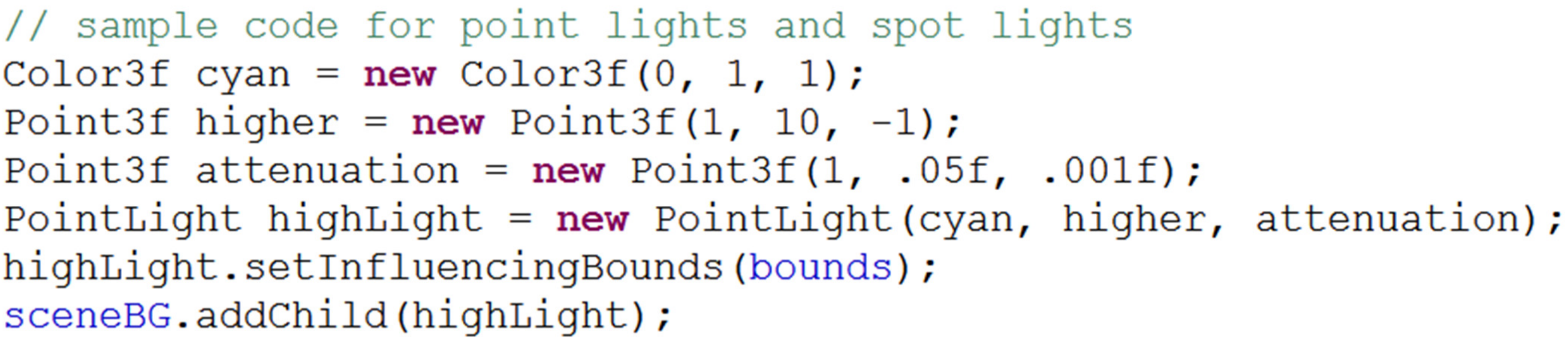


- has a color
- has an attenuation, intensity decreases as distance from light increases

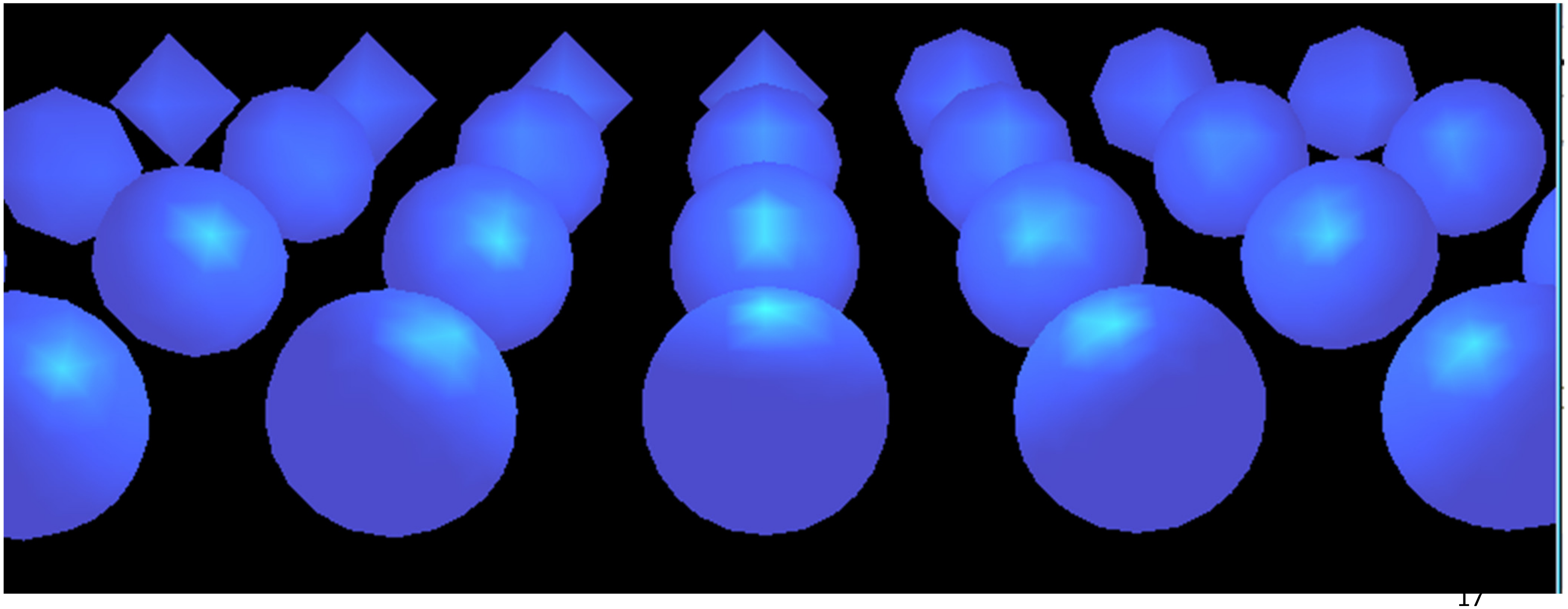
Point Light

- Attenuation of Point Light has three factors:
  - constant attenuation, ac
  - linear attenuation, la
  - quadratic attenuation, qa
- light intensity at a given point distance d away from the light =
  1.0 / (ac + la * d + qa * d * d)

Point Light

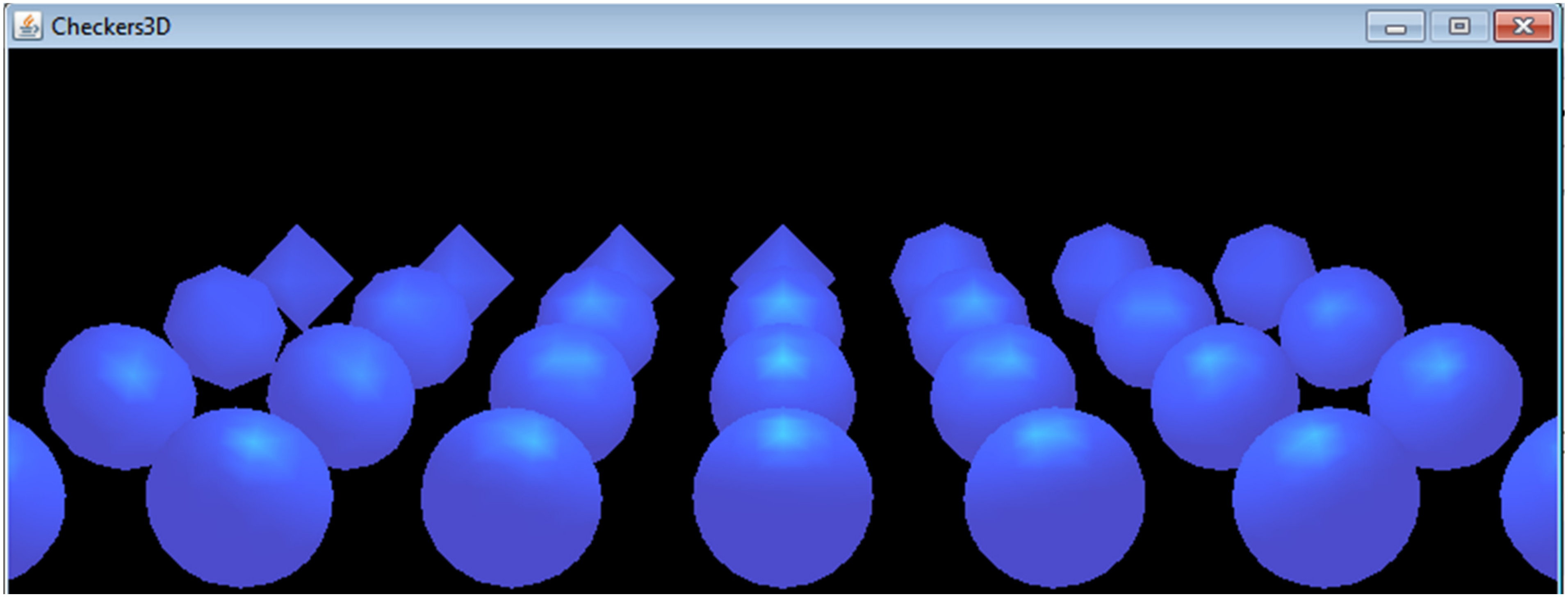


```
// sample code for point lights and spot lights
Color3f cyan = new Color3f(0, 1, 1);
Point3f higher = new Point3f(1, 10, -1);
Point3f attenuation = new Point3f(1, .05f, .001f);
PointLight highLight = new PointLight(cyan, higher, attenuation);
highLight.setInfluencingBounds(bounds);
sceneBG.addChild(highLight);
```



Point Light

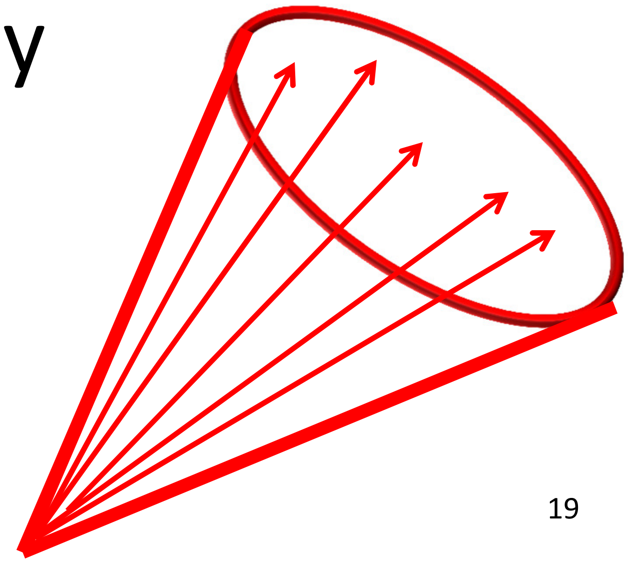
- Change y coordinate of location from 10 to 20



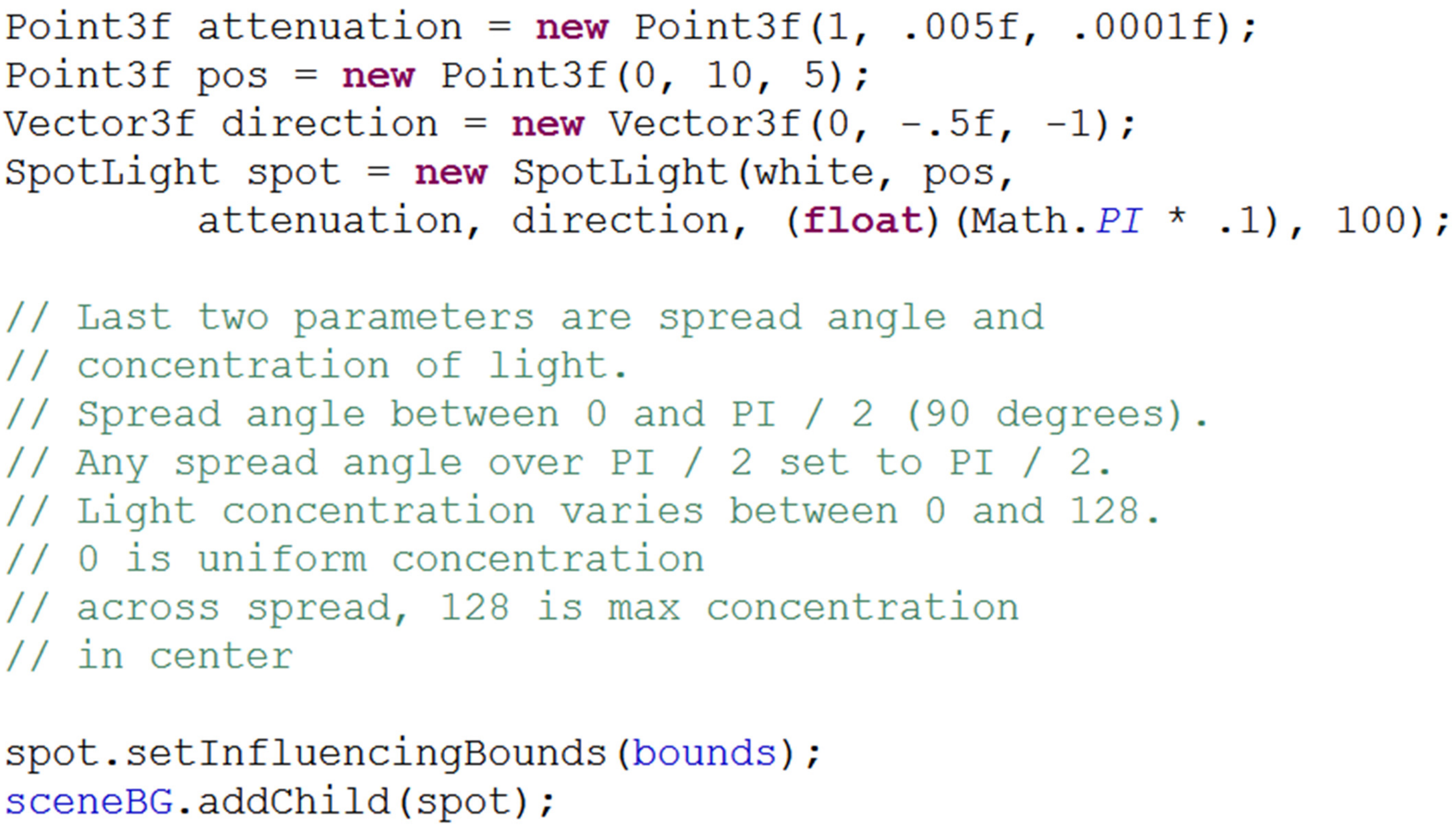
- We see the things the light interacts with

Spotlight

- Similar to Point Light
- has location and direction, but does not emit light in all directions
- emits light in a cone shape region
- attenuation like Point Light, but add attenuation for light rays away from central direction

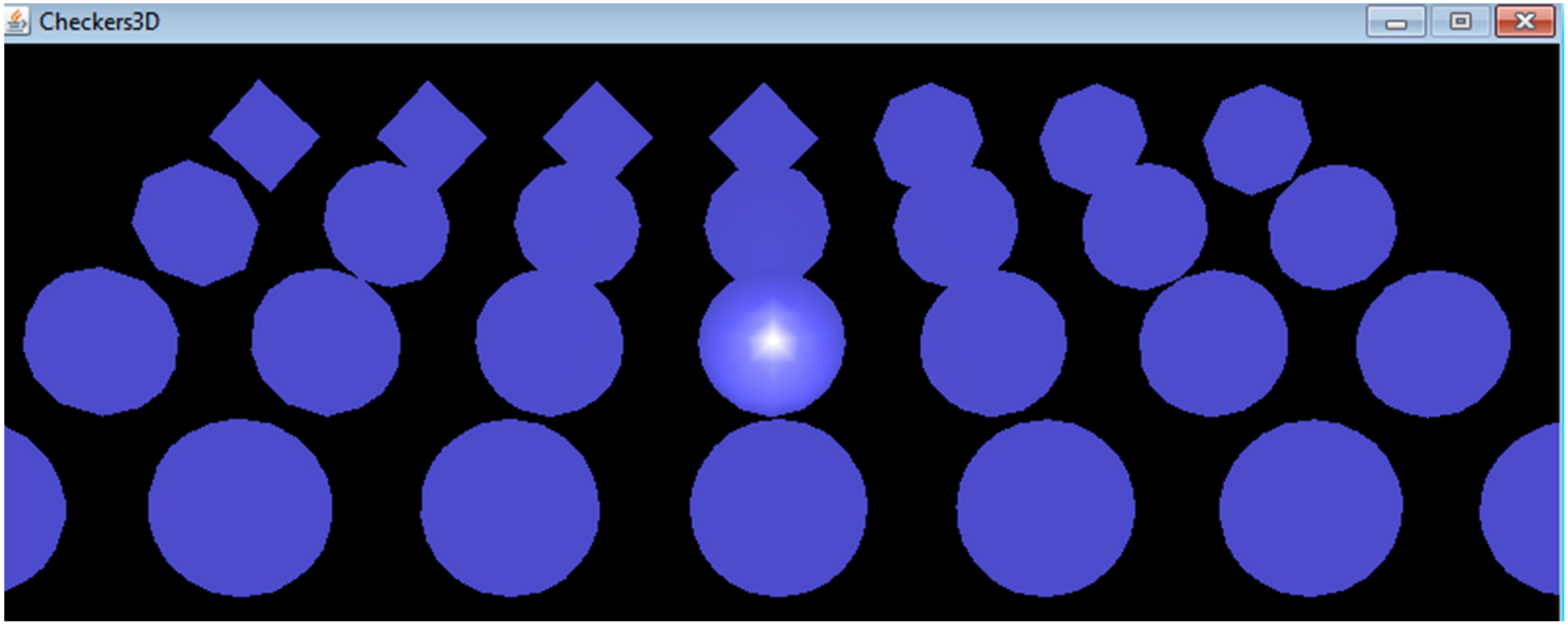


Spotlight

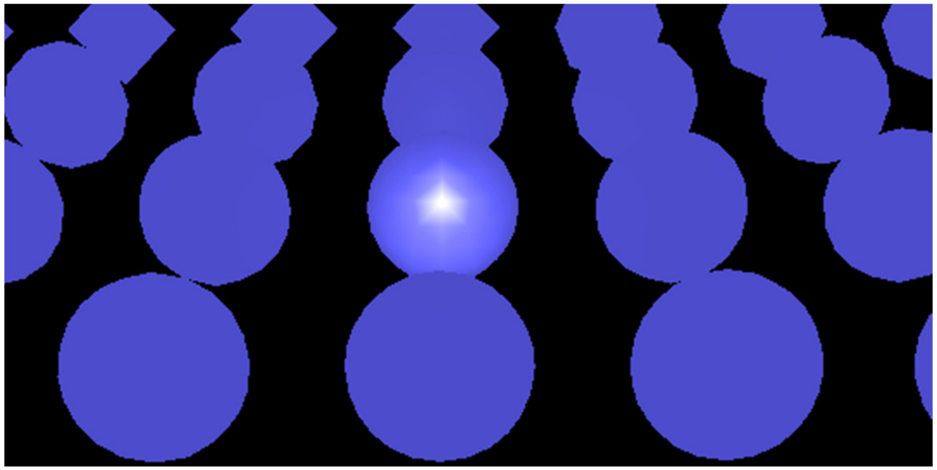


```
Point3f attenuation = new Point3f(1, .005f, .0001f);
Point3f pos = new Point3f(0, 10, 5);
Vector3f direction = new Vector3f(0, -.5f, -1);
SpotLight spot = new SpotLight(white, pos,
        attenuation, direction, (float)(Math.PI * .1), 100);

// Last two parameters are spread angle and
// concentration of light.
// Spread angle between 0 and PI / 2 (90 degrees).
// Any spread angle over PI / 2 set to PI / 2.
// Light concentration varies between 0 and 128.
// 0 is uniform concentration
// across spread, 128 is max concentration
// in center

spot.setInfluencingBounds(bounds);
sceneBG.addChild(spot);
```

Spotlight Effects

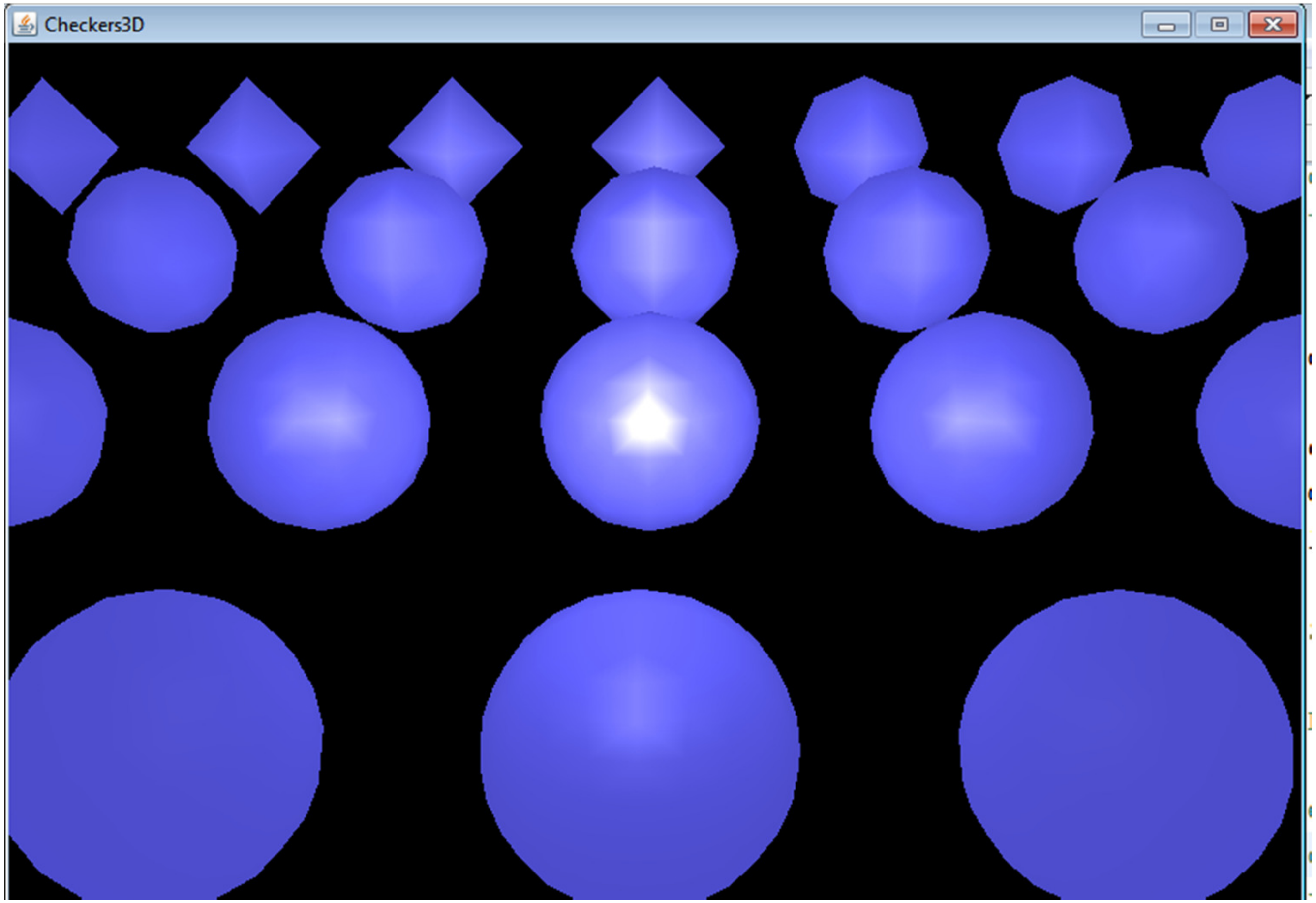


- attenuation changed to PI / 2



Spotlight Effects

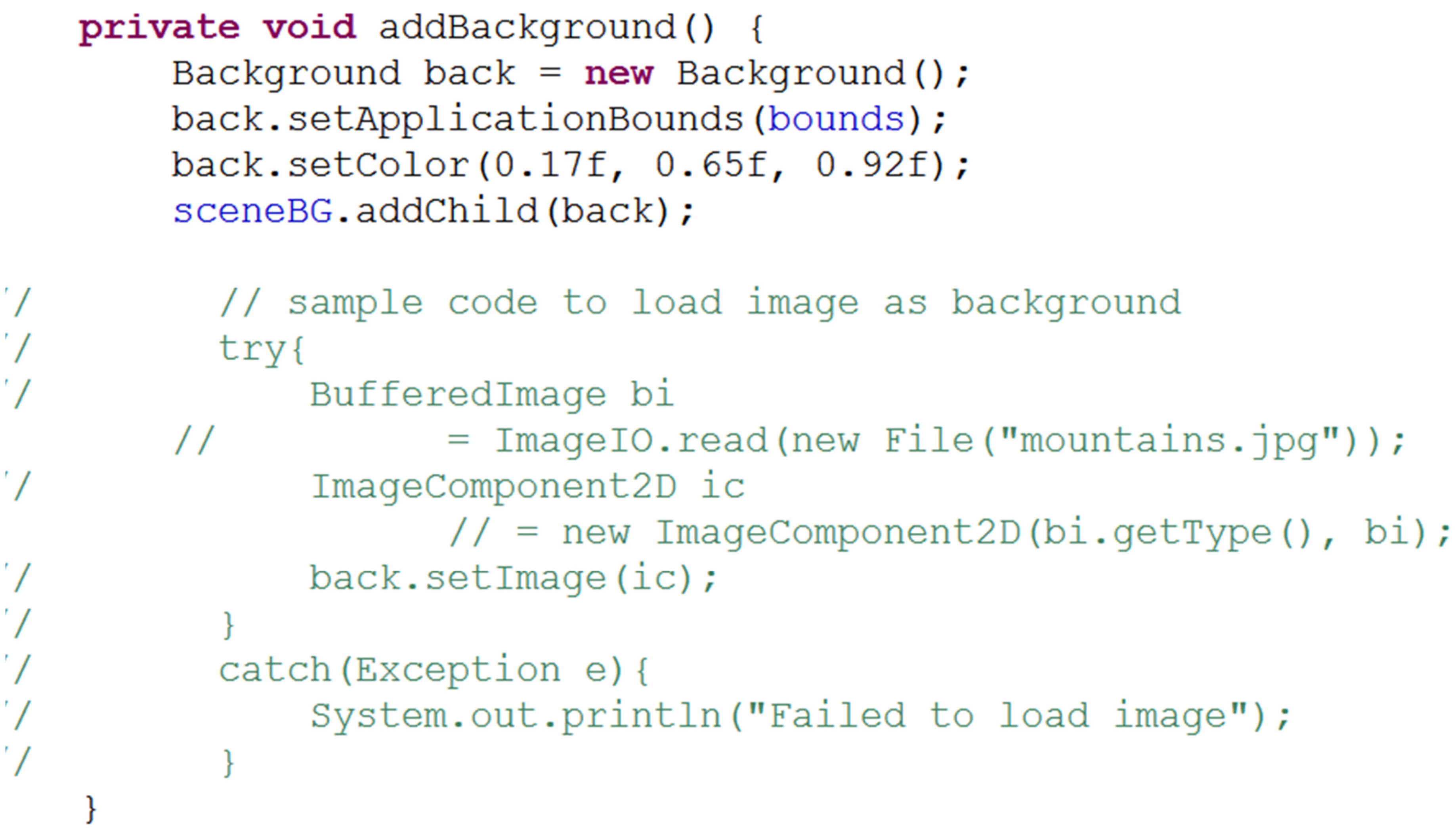
- Concentration changed from 100 to 10, spread angle still set at PI / 2



Background

- can add a background image or color
- draw at the back of a scene and is not altered by camera movement in the scene

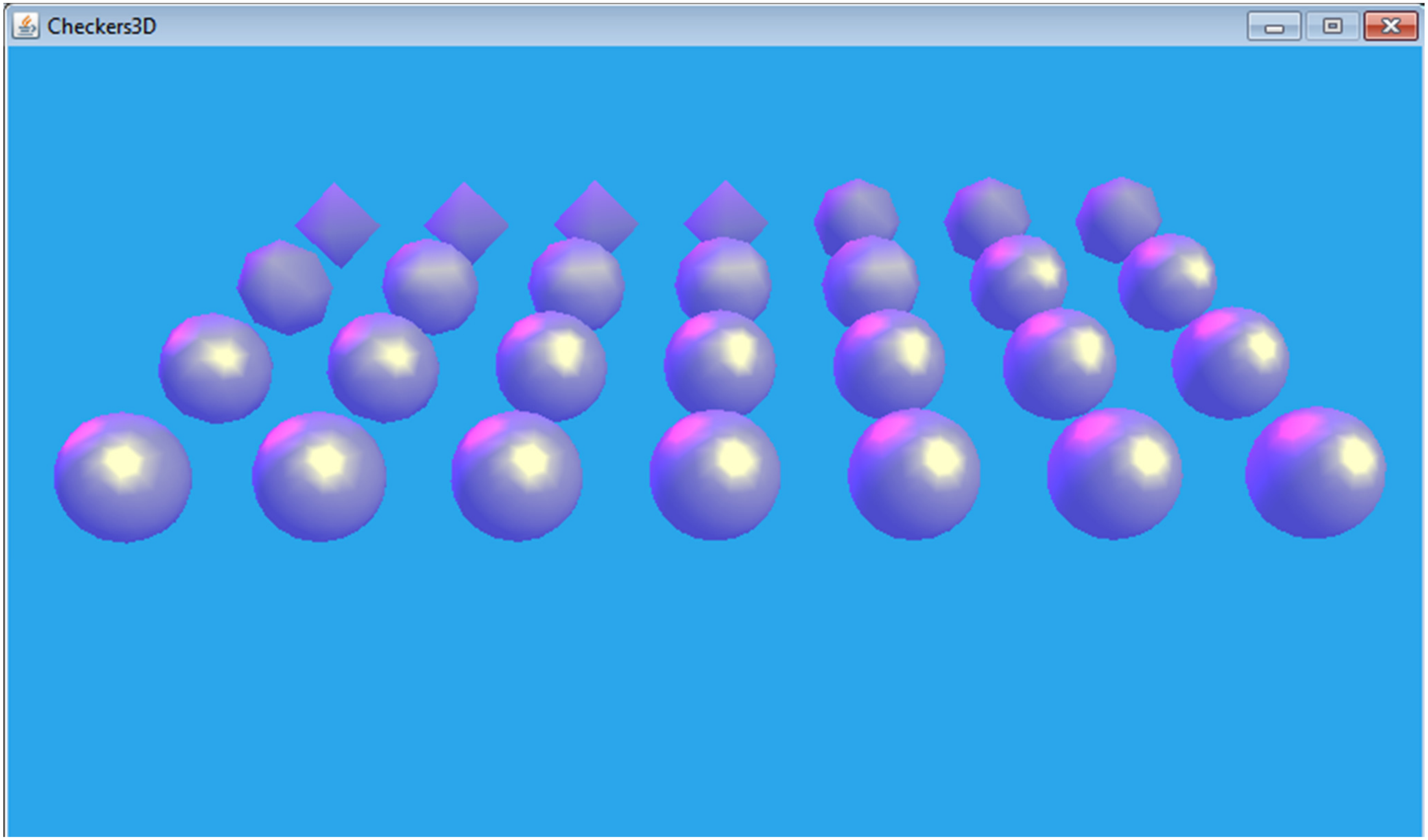
Background Code



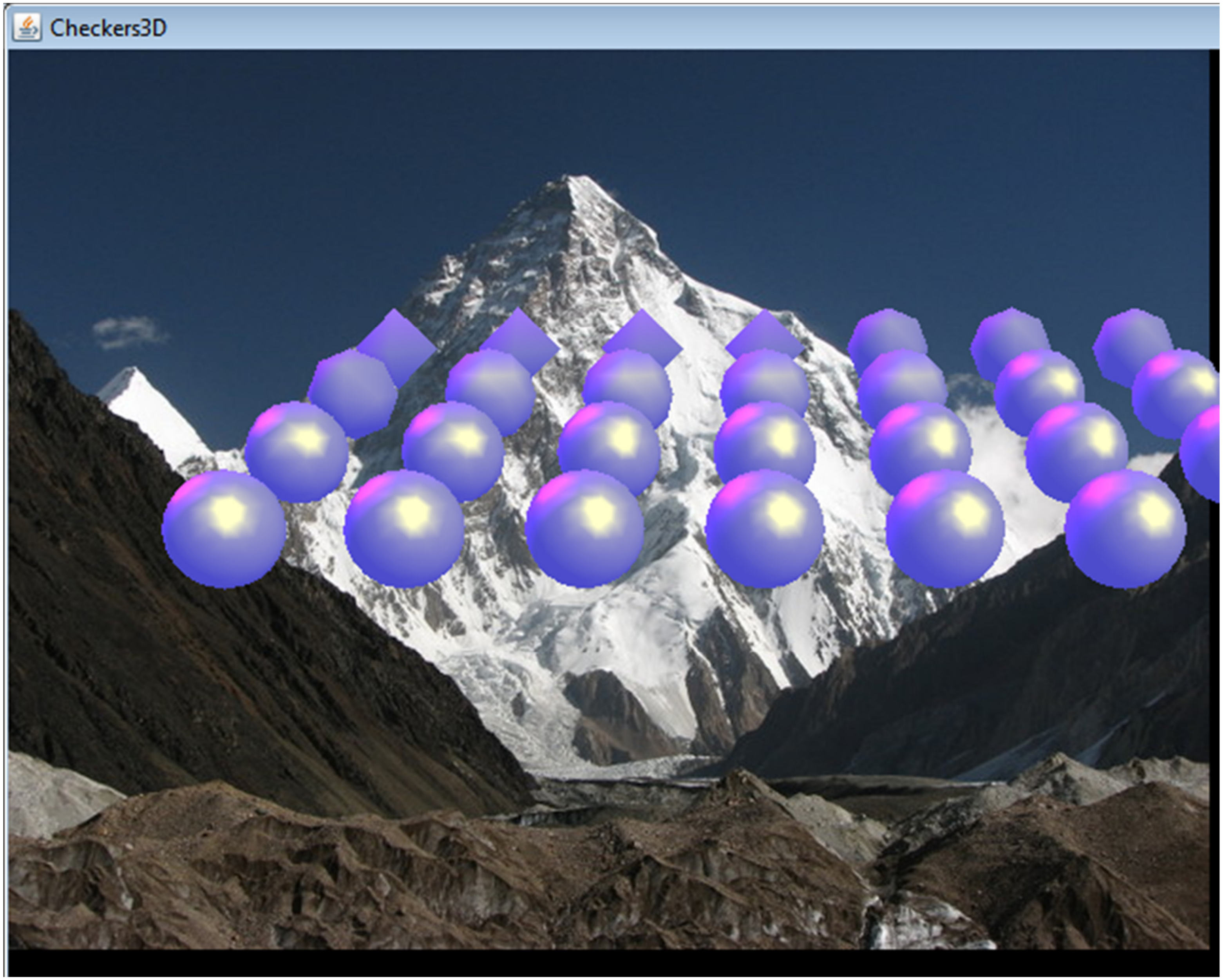
```
    private void addBackground() {
        Background back = new Background();
        back.setApplicationBounds(bounds);
        back.setColor(0.17f, 0.65f, 0.92f);
        sceneBG.addChild(back);

'/          // sample code to load image as background
'/          try{
'/              BufferedImage bi
        //              = ImageIO.read(new File("mountains.jpg"));
'/              ImageComponent2D ic
                    // = new ImageComponent2D(bi.getType(), bi);
'/              back.setImage(ic);
'/          }
'/          catch(Exception e){
'/              System.out.println("Failed to load image");
'/          }
    }
```

Background Effect - Color

- ambient and directional lights



Background Effect - Image



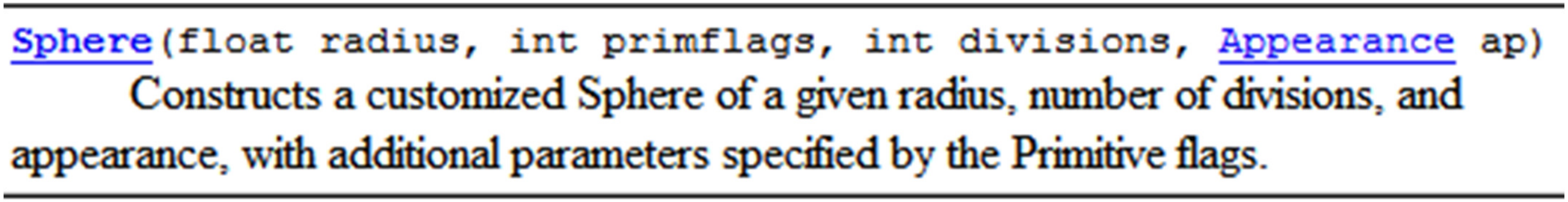
Adding Shapes

- Documentation for Java3D package at http://download.java.net/media/java3d/javadoc/1.5.1/index.html
- Add visible objects to a scene requires adding *Primitives* or creating a class that extends *Shape3D*
- built in subclasses of primitive include:
  - box, cone, cylinder, sphere

Adding Spheres

- Multiple Sphere constructors but all have some variation of these parameters
  - radius, size of sphere
  - primFlags, A number of constants that affect how the sphere is created (for example should the appearance be allowed to be changed)
  - divisions, affects the number of polygons used to construct the sphere (divisions != total number)
  - appearance, how the sphere should look

Sphere Constructor



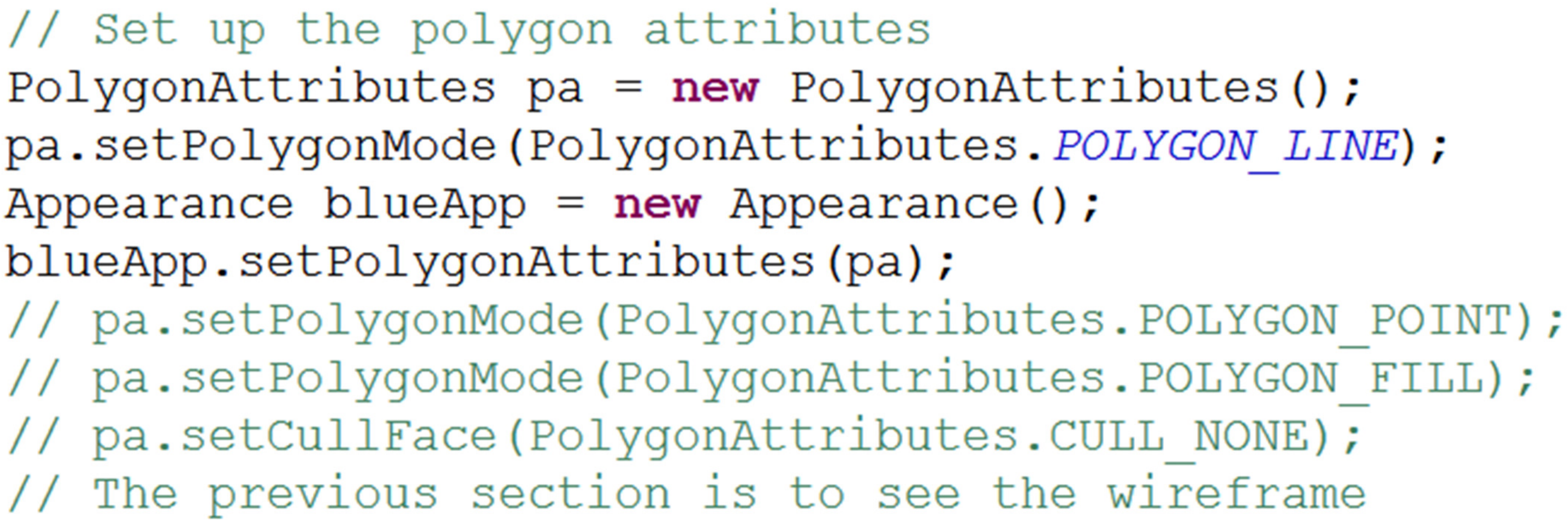
```
Sphere(float radius, int primflags, int divisions, Appearance ap)
```

Constructs a customized Sphere of a given radius, number of divisions, and appearance, with additional parameters specified by the Primitive flags.

- primflags - refer to Sphere class for options, GENERATE_NORMALS in our case
- radius = 2.0f
- vary divisions from 4 to 31
- Appearance, show wireframe and blue materials

Appearances

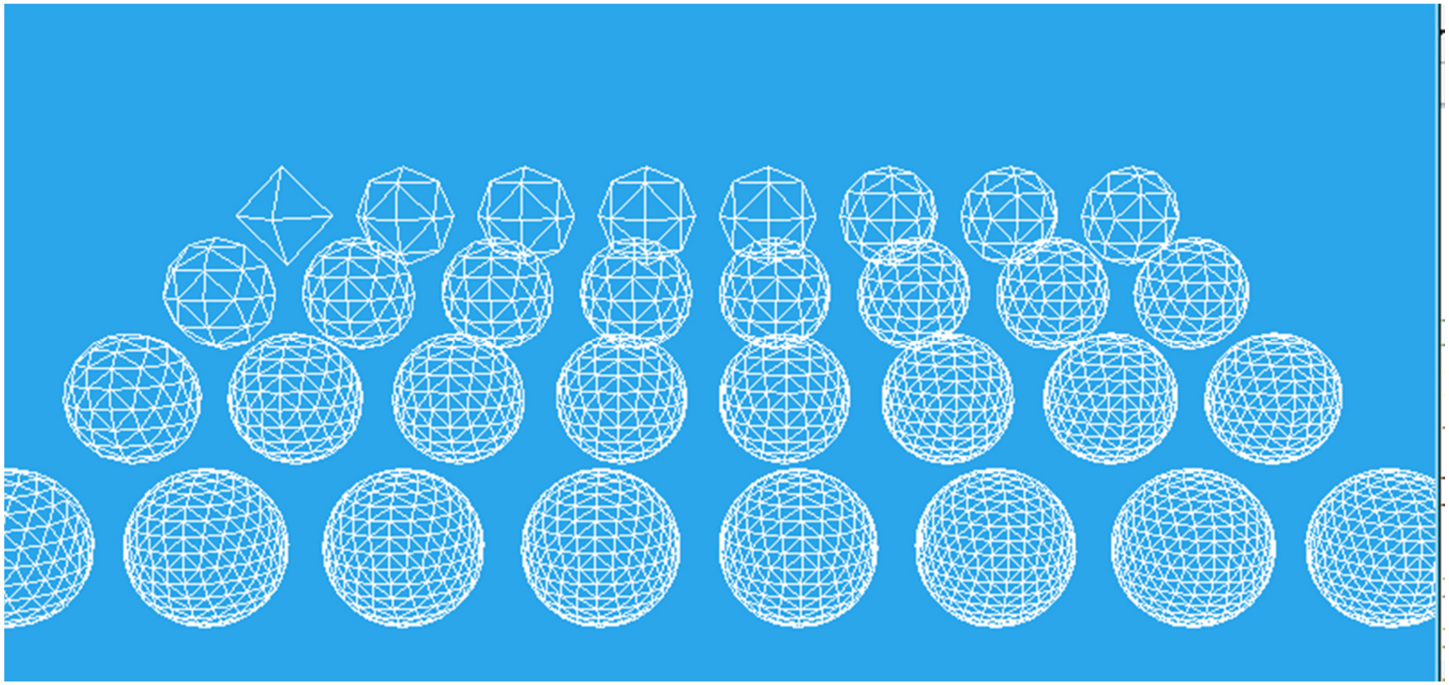
- Program includes two hard coded appearances for the spheres
- First, just show the polygons



```
// Set up the polygon attributes
PolygonAttributes pa = new PolygonAttributes();
pa.setPolygonMode(PolygonAttributes.POLYGON_LINE);
Appearance blueApp = new Appearance();
blueApp.setPolygonAttributes(pa);
// pa.setPolygonMode(PolygonAttributes.POLYGON_POINT);
// pa.setPolygonMode(PolygonAttributes.POLYGON_FILL);
// pa.setCullFace(PolygonAttributes.CULL_NONE);
// The previous section is to see the wireframe
```

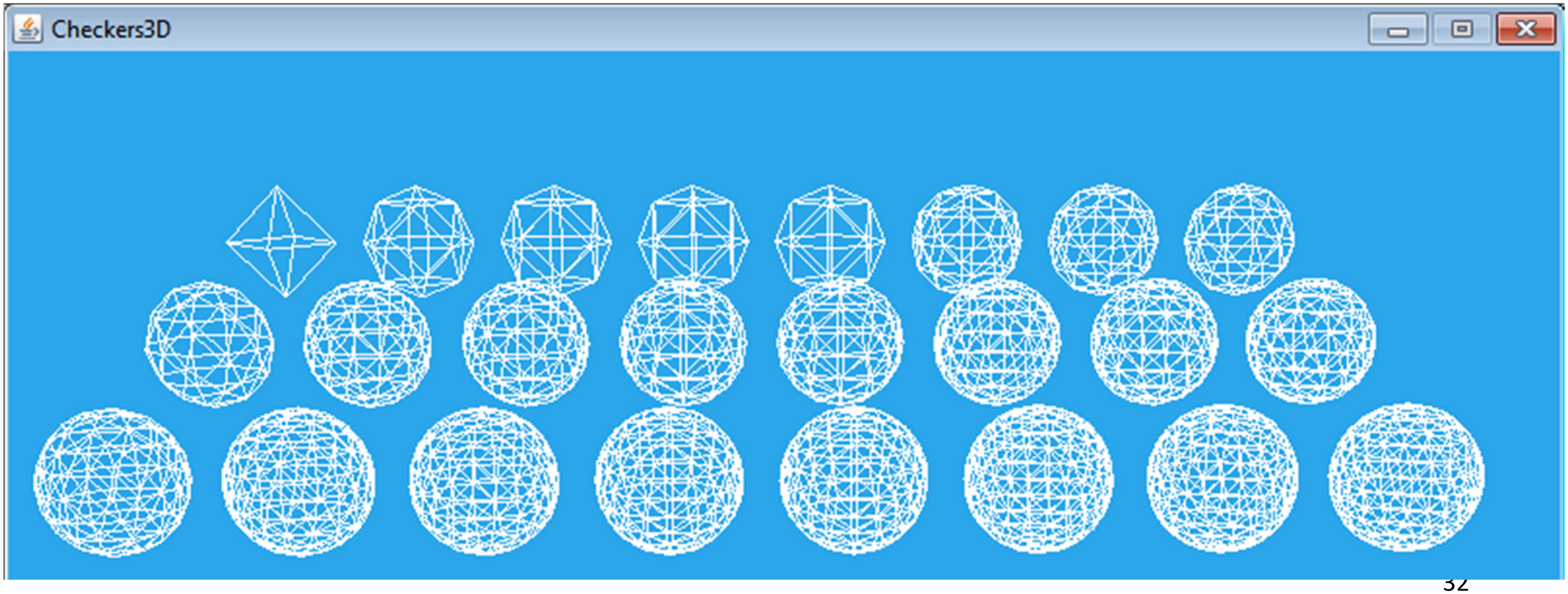
Wireframes

- By default inside faces are culled, not visible
- notice difference in divisions
  - why not just crank divisions up to 100s?



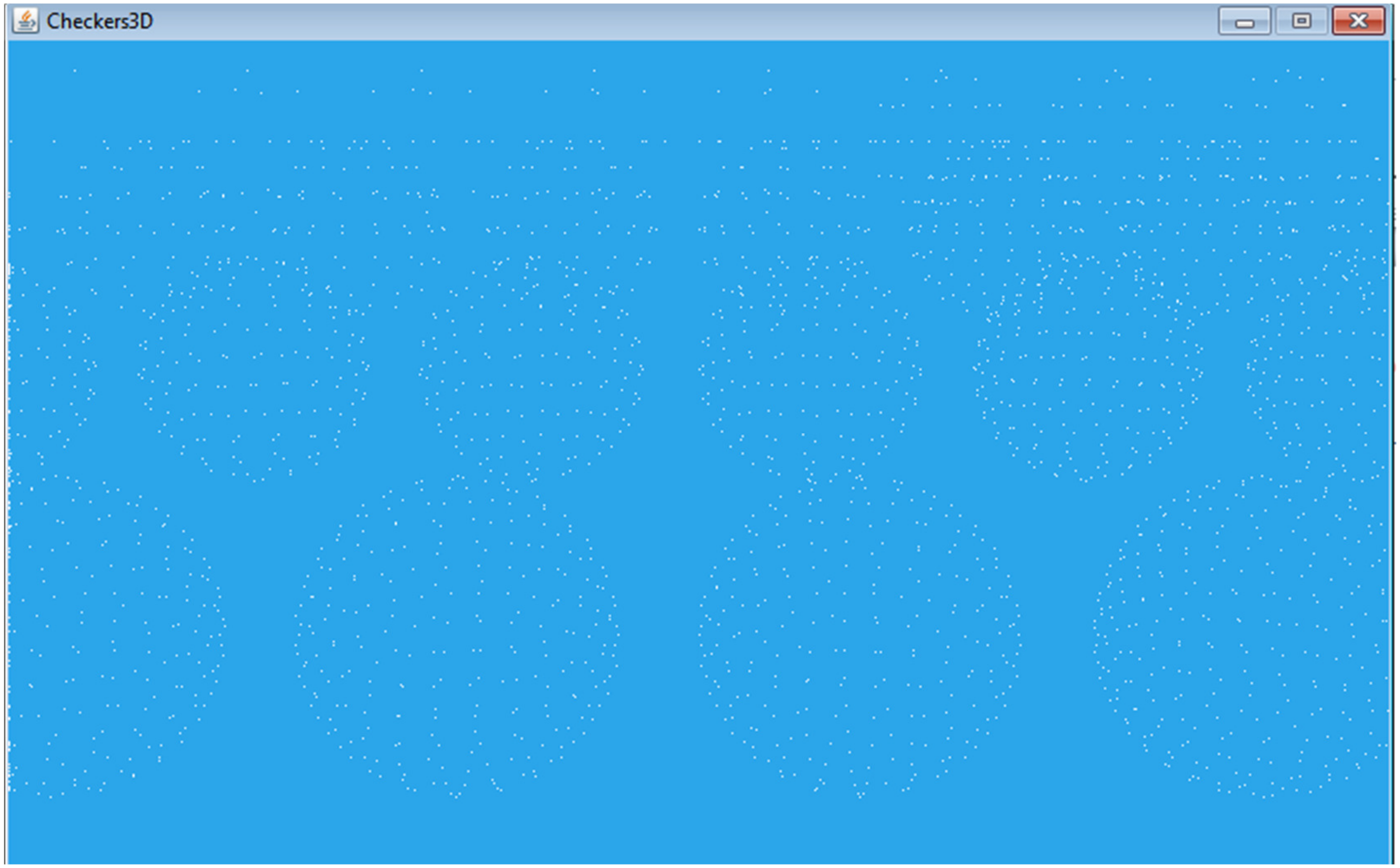
Wireframes

- CULL_NONE
- Notice difference in smaller divisions
- we see lines the "backs" of lines



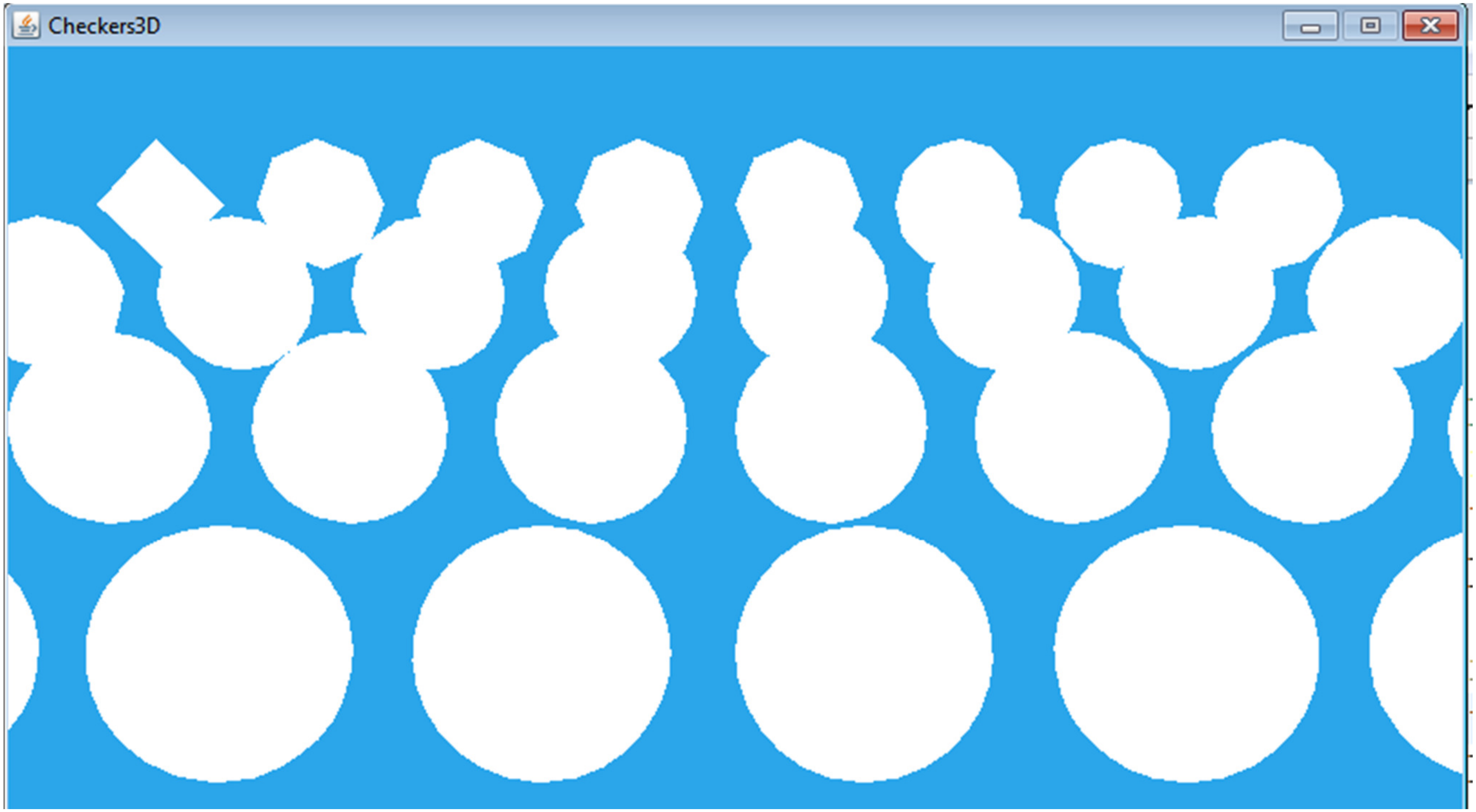
Wireframes

- Using POLYGON_POINT instead of POLYGON_LINE



Polygon Attributes

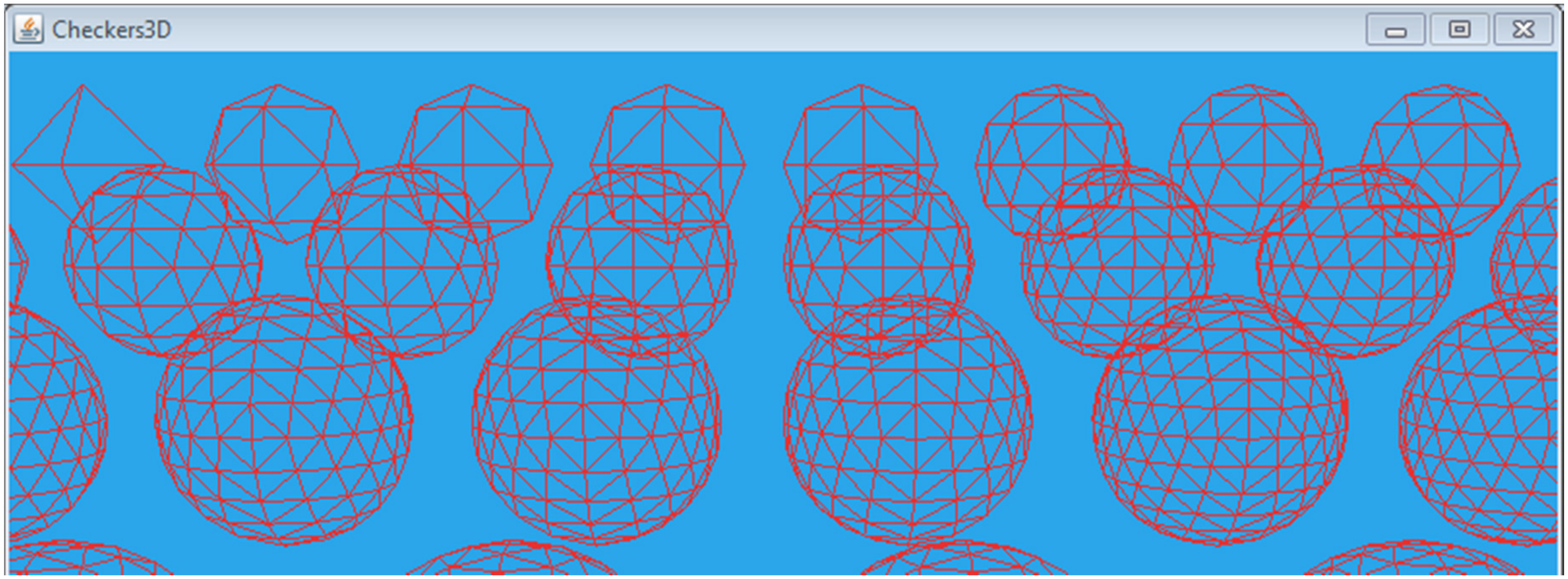
- Using POLYGON_FILL



Changing Color

- Color attribute of an appearance can be changed, default is white

```
Appearance blueApp = new Appearance();
ColoringAttributes ca = new ColoringAttributes();
ca.setColor(new Color3f(.9f, .2f, .2f));
blueApp.setColoringAttributes(ca);
```

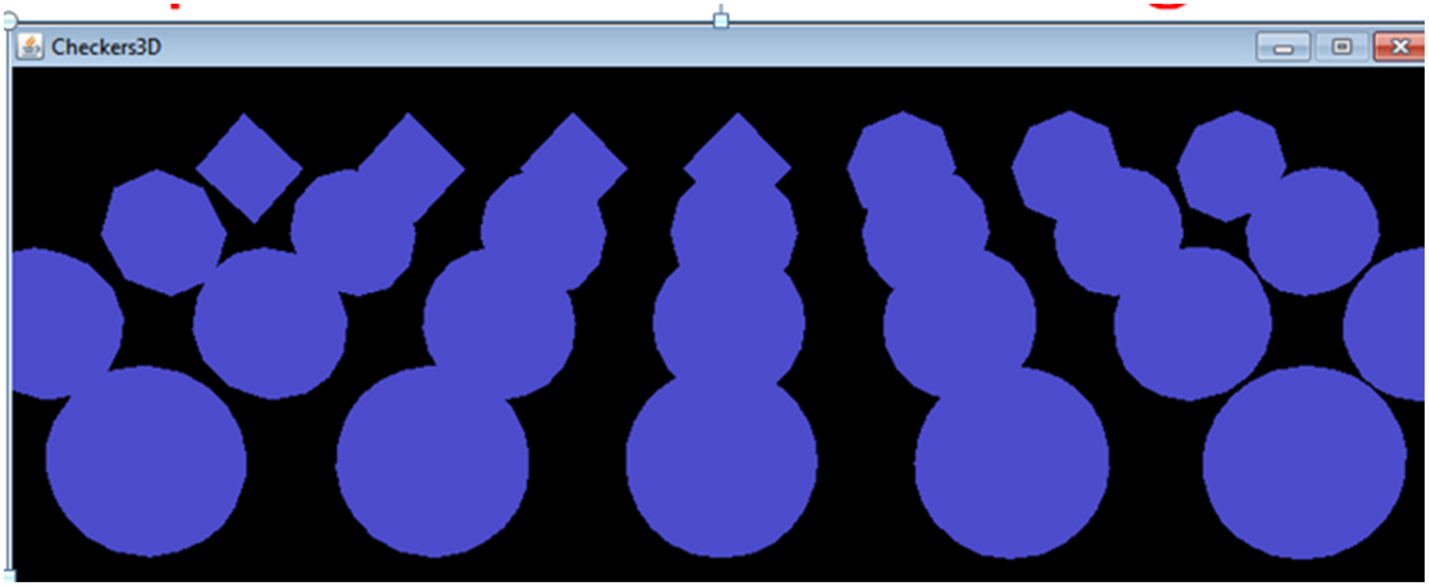


Materials

- More realistic appearances created using materials
- Material class
- specify four colors and a value for shininess (1 to 128)
- four colors for ambient, emissive, diffuse, and specular properties of the material
- Define how light interacts with the material and the light it gives off.

Material Properties

- Ambient Color: how much (and what color) ambient light is reflected by the material
- Recall when only ambient light in the world

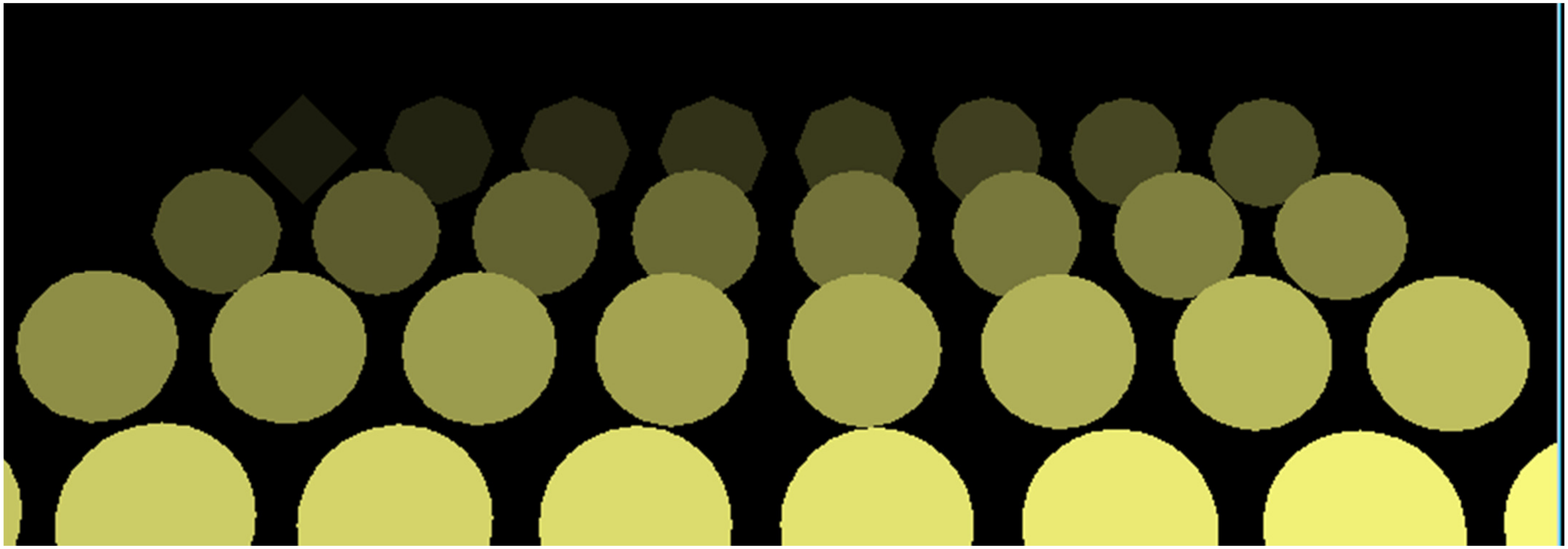


Material Properties

- Emissive Light
  - the color of light the material gives off itself
  - material glows, but does not illuminate other materials
  - to create a flashlight pair a spotlight and a shape with a material that gives off light
  - examples so far emissive light for spheres was black (none)
  - dim the lights and let the spheres give off a bright yellow light

Emissive Light

- No other lights

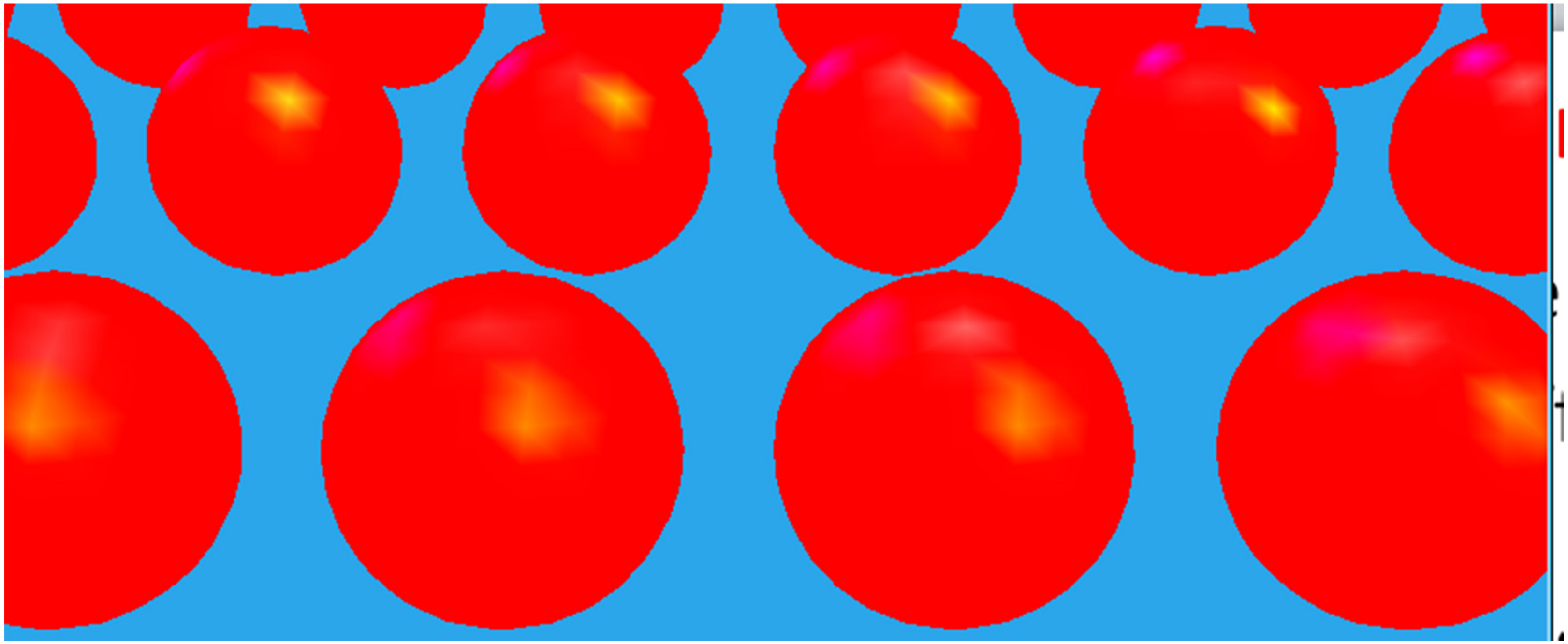


Diffuse and Specular Properties

- Ambient color is response to ambient light which appears to come from all directions
- diffuse color reflects light coming from one direction (directional, point, and spot lights)
  - angle in != angle out
  - various angles of reflection
- specular color, follows law of reflection
  - angle in = angle out

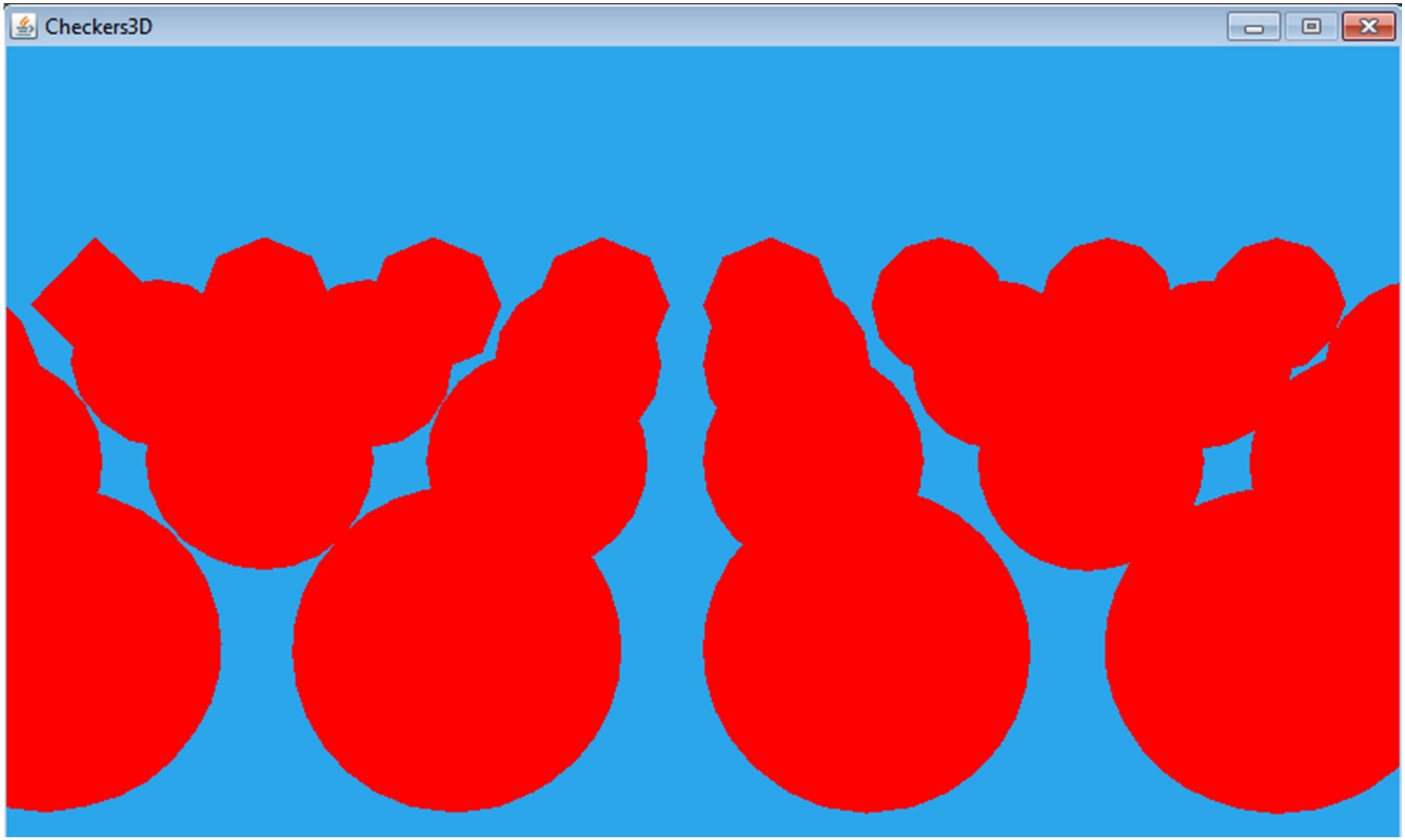
Materials

- Billiard Ball like material
- ambient emissive diffuse specular shininess
- (red, black, red, white, 70)
- notice glinting off edges



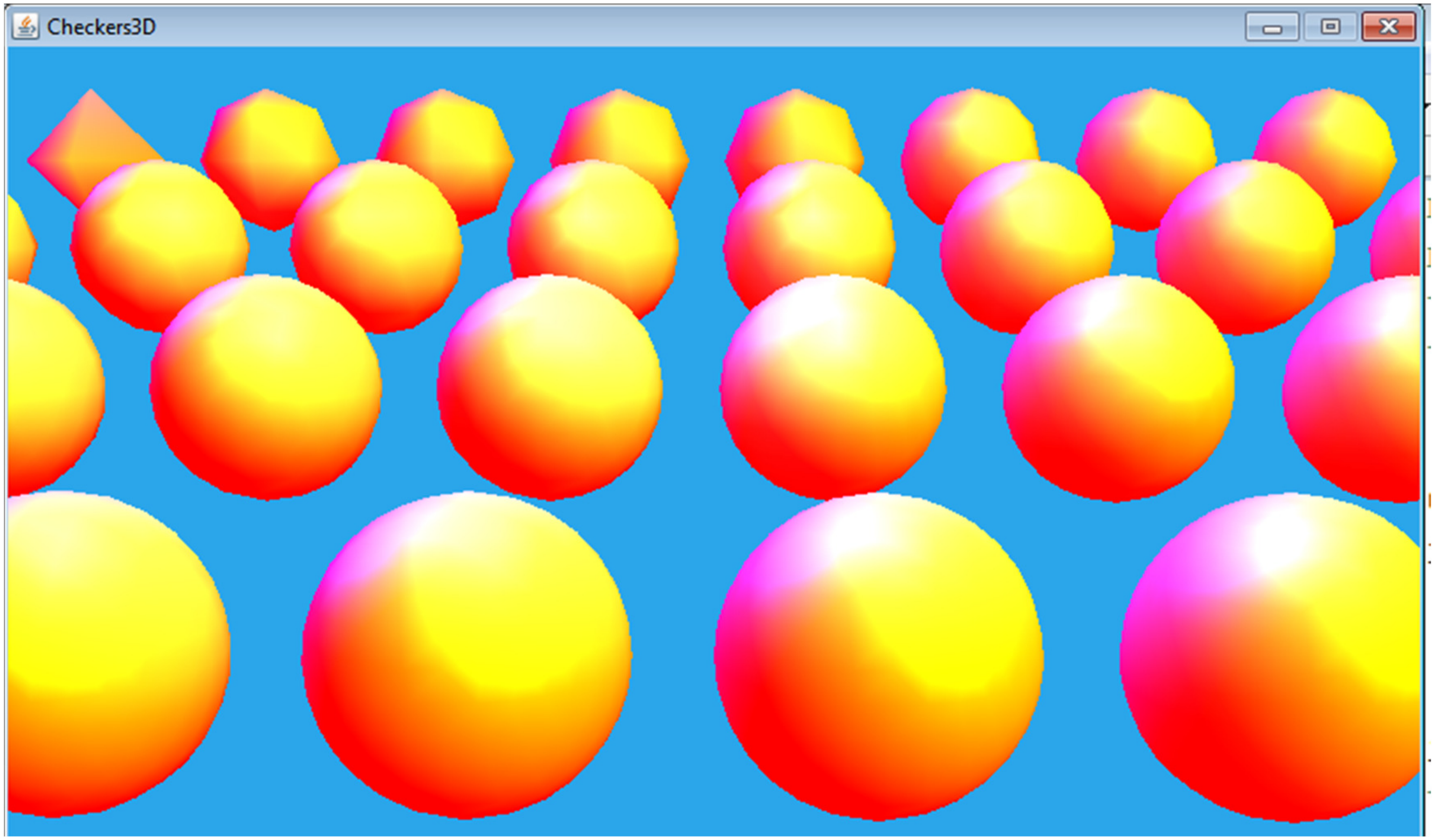
Materials

- Specular set to black



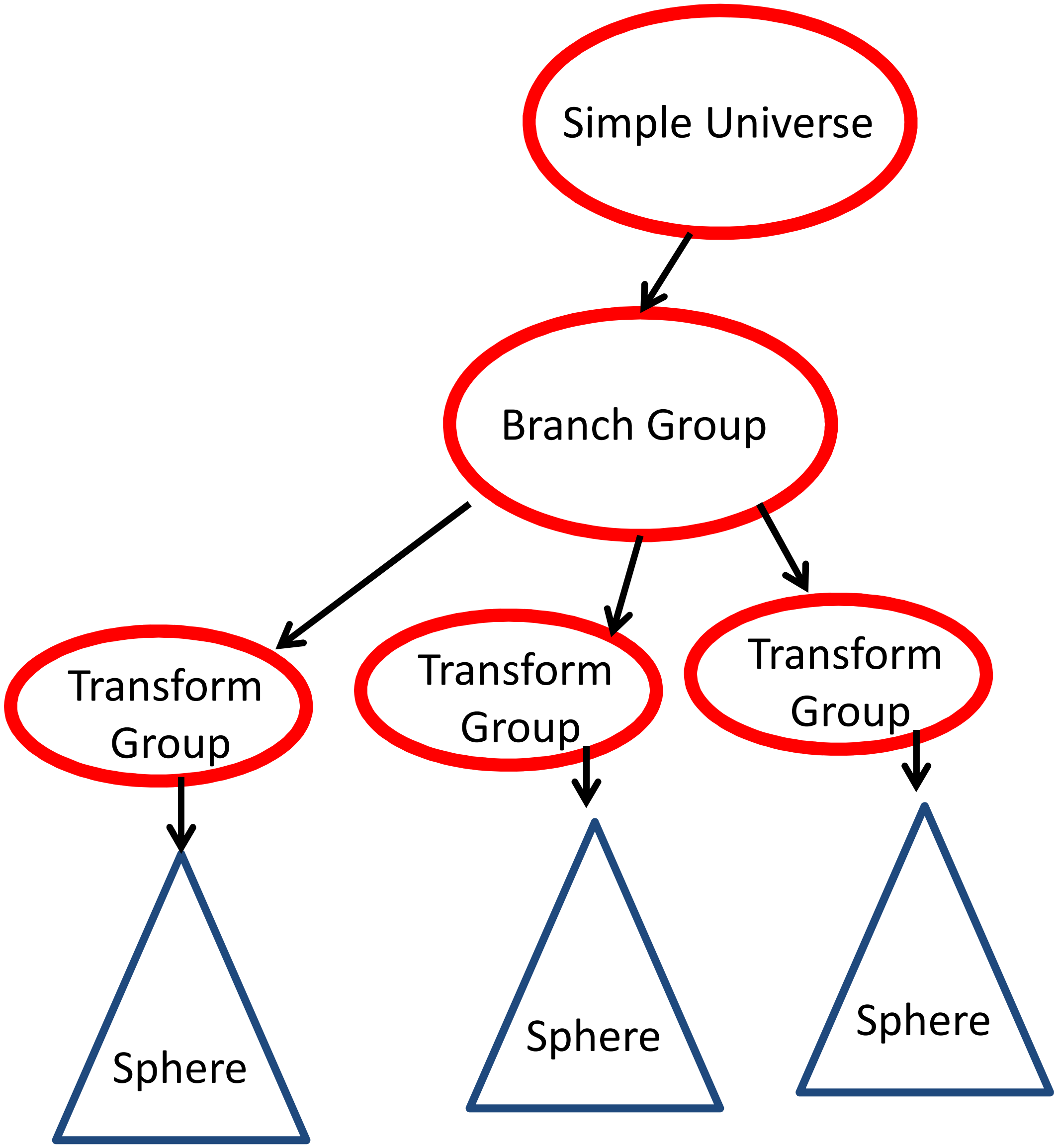
Materials

- ambient emissive diffuse specular shininess
- (red, black, white, white, 25)

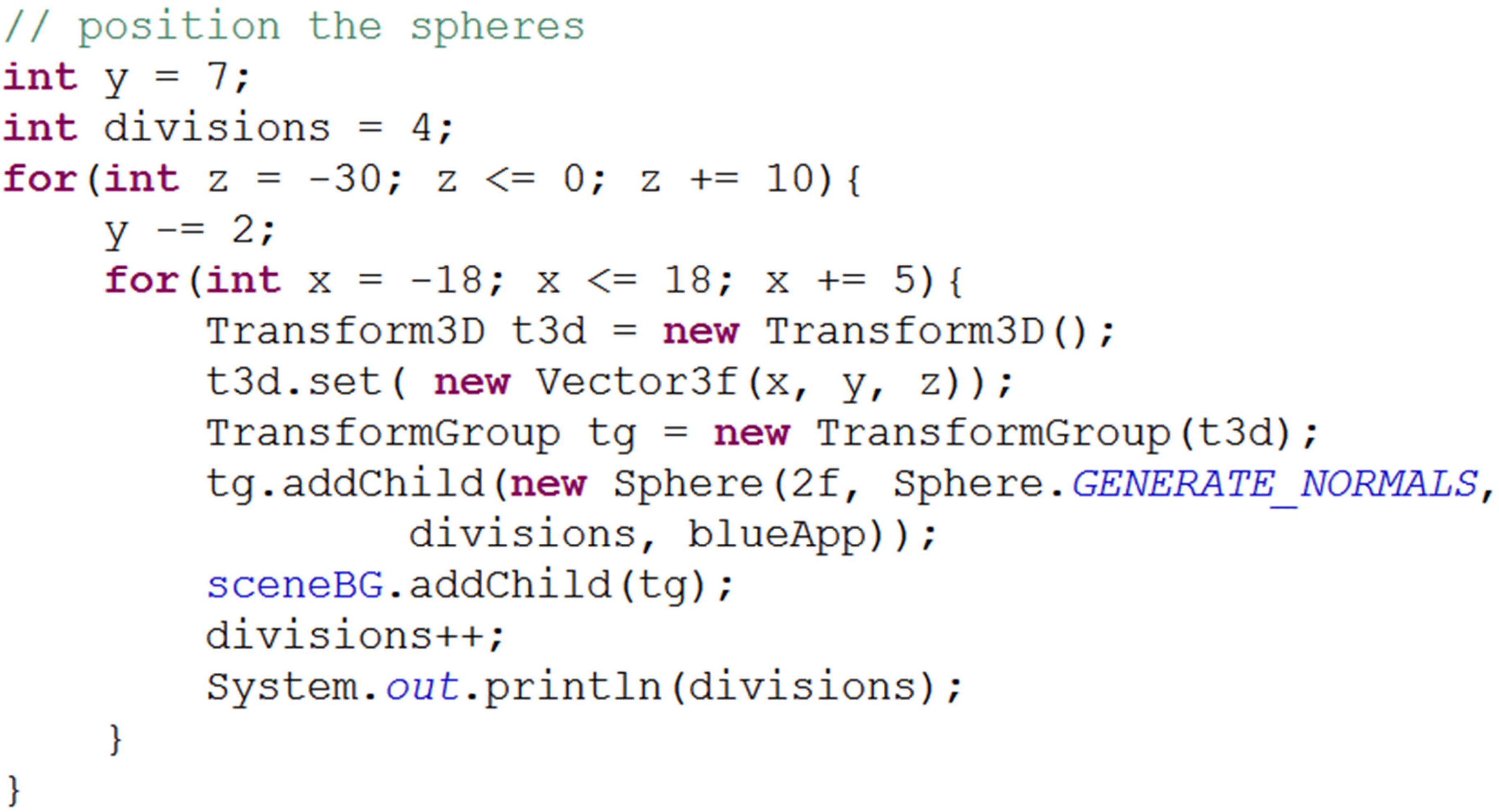


Positioning Spheres

- To position spheres each has its own Transform Group
- change x, y, and z for each transform group
- can't just update one TG, otherwise position changes

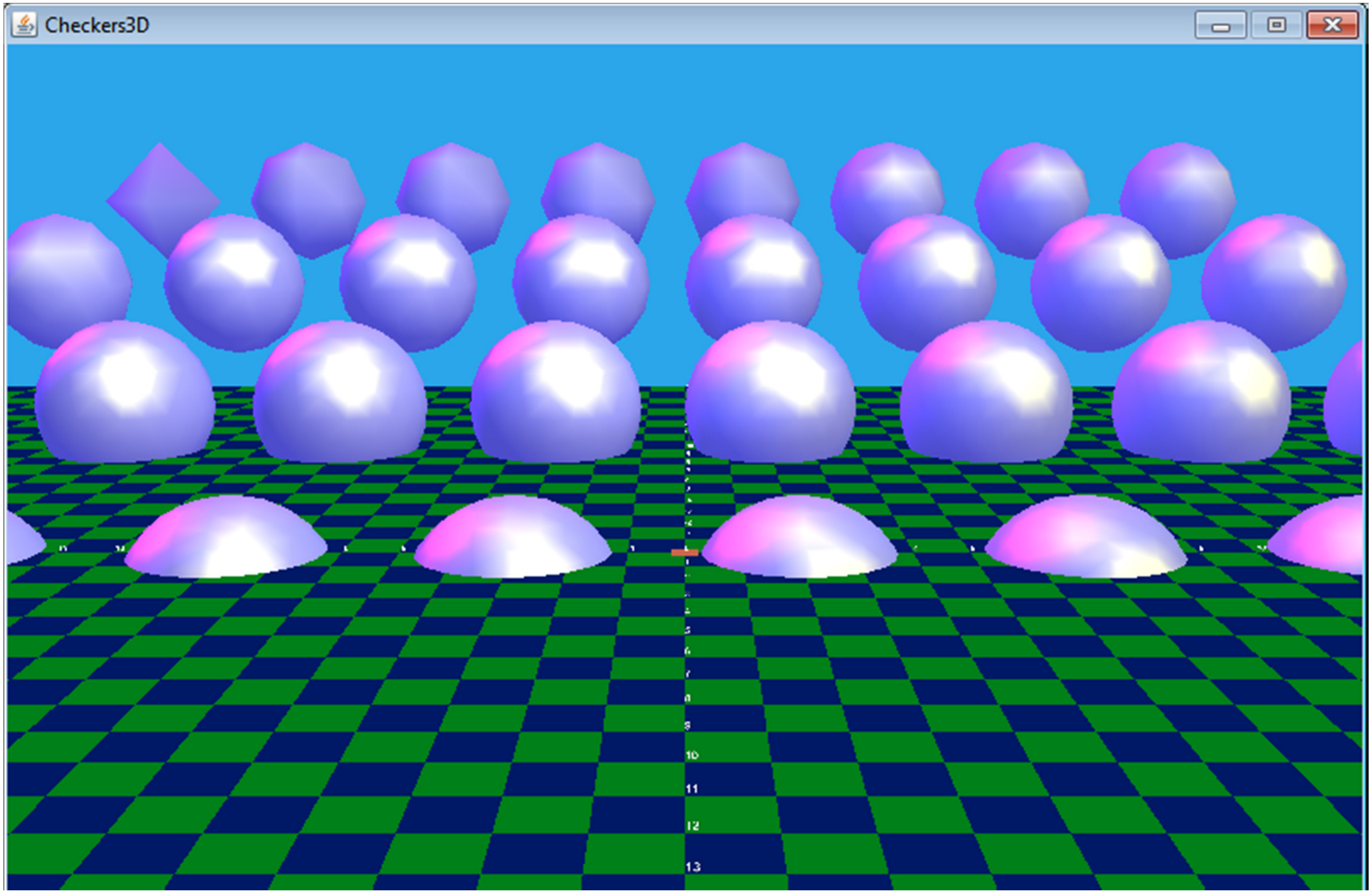


Positioning Spheres



```java
// position the spheres
int y = 7;
int divisions = 4;
for(int z = -30; z <= 0; z += 10){
    y -= 2;
    for(int x = -18; x <= 18; x += 5){
        Transform3D t3d = new Transform3D();
        t3d.set( new Vector3f(x, y, z));
        TransformGroup tg = new TransformGroup(t3d);
        tg.addChild(new Sphere(2f, Sphere.GENERATE_NORMALS,
                divisions, blueApp));
        sceneBG.addChild(tg);
        divisions++;
        System.out.println(divisions);
    }
}
```

Adding the Checker Floor



CheckerFloor

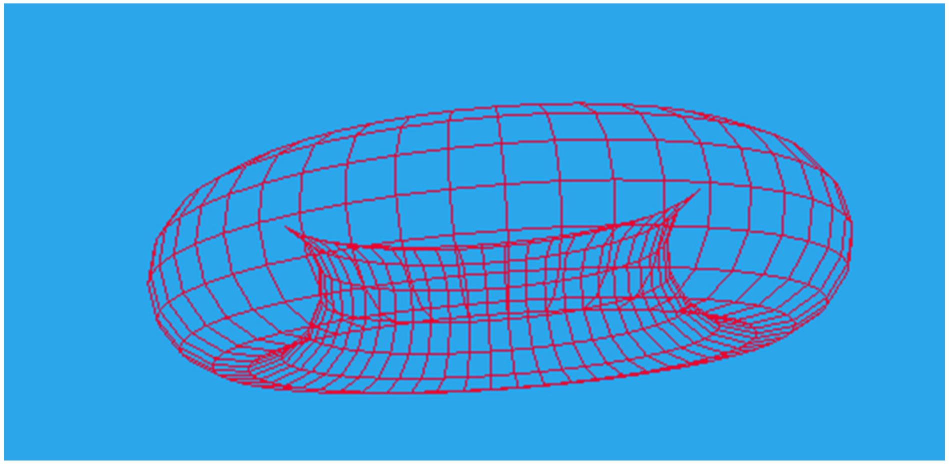
- Not a standard Java3D class
- creates its own branch group
- consists of
  - 2 sets of colored tiles, blue and green
  - a red colored tile at the center
  - labels which are Text2D objects

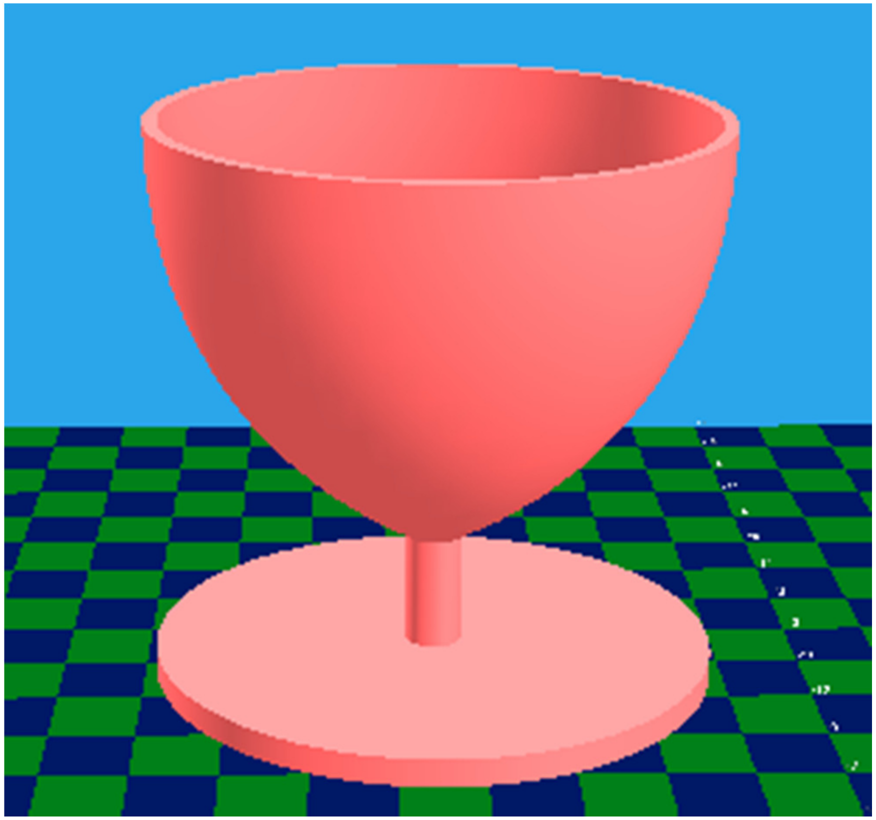
Colored Tiles

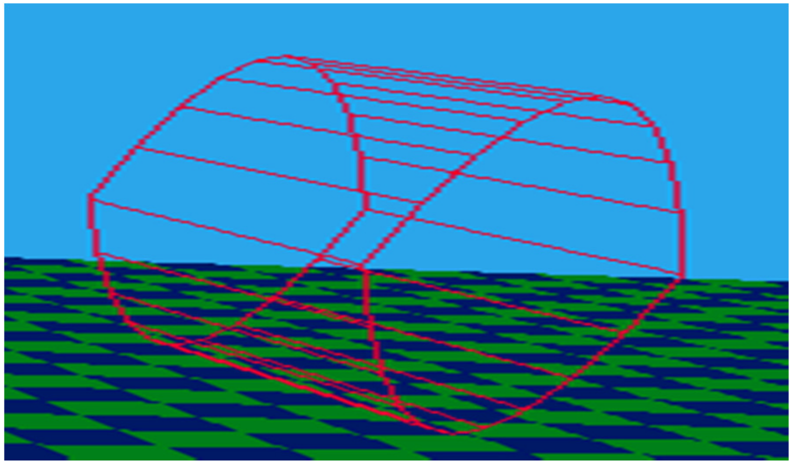
- ColoredTiles class extends built in Shjape3D class
- Uses a QuadArray to represent the tiles
- QuadArray a built in Java3D class
  - stores sets of 4 points that define individual quadrilaterals
  - In this case a flat surface, but quads do not have to be co-planar
  - quads don't have to be connected to each other

Morph and Lathe

- Build shapes by defining an outline and rotating outline 360 degrees

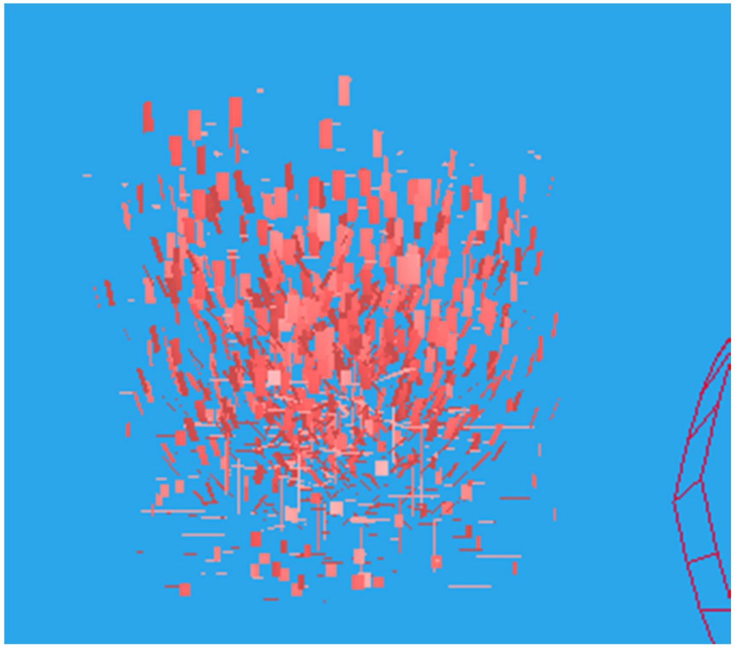


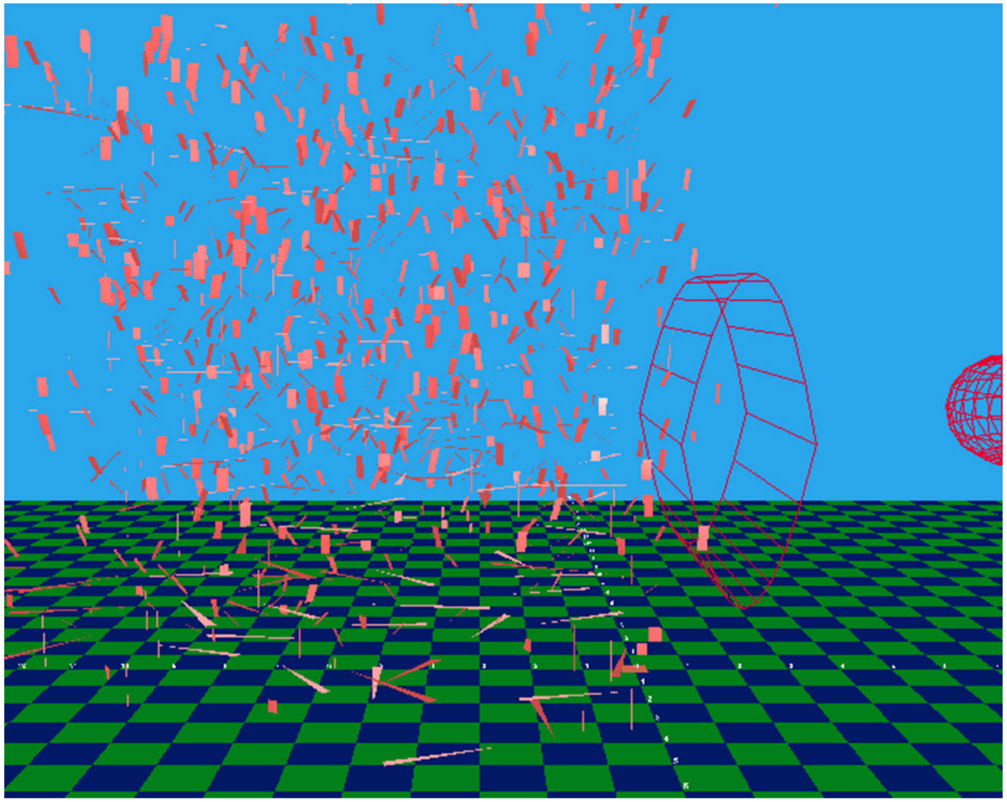




Exploding a Shape3D

- can alter coordinates of quads to create an explosion affect

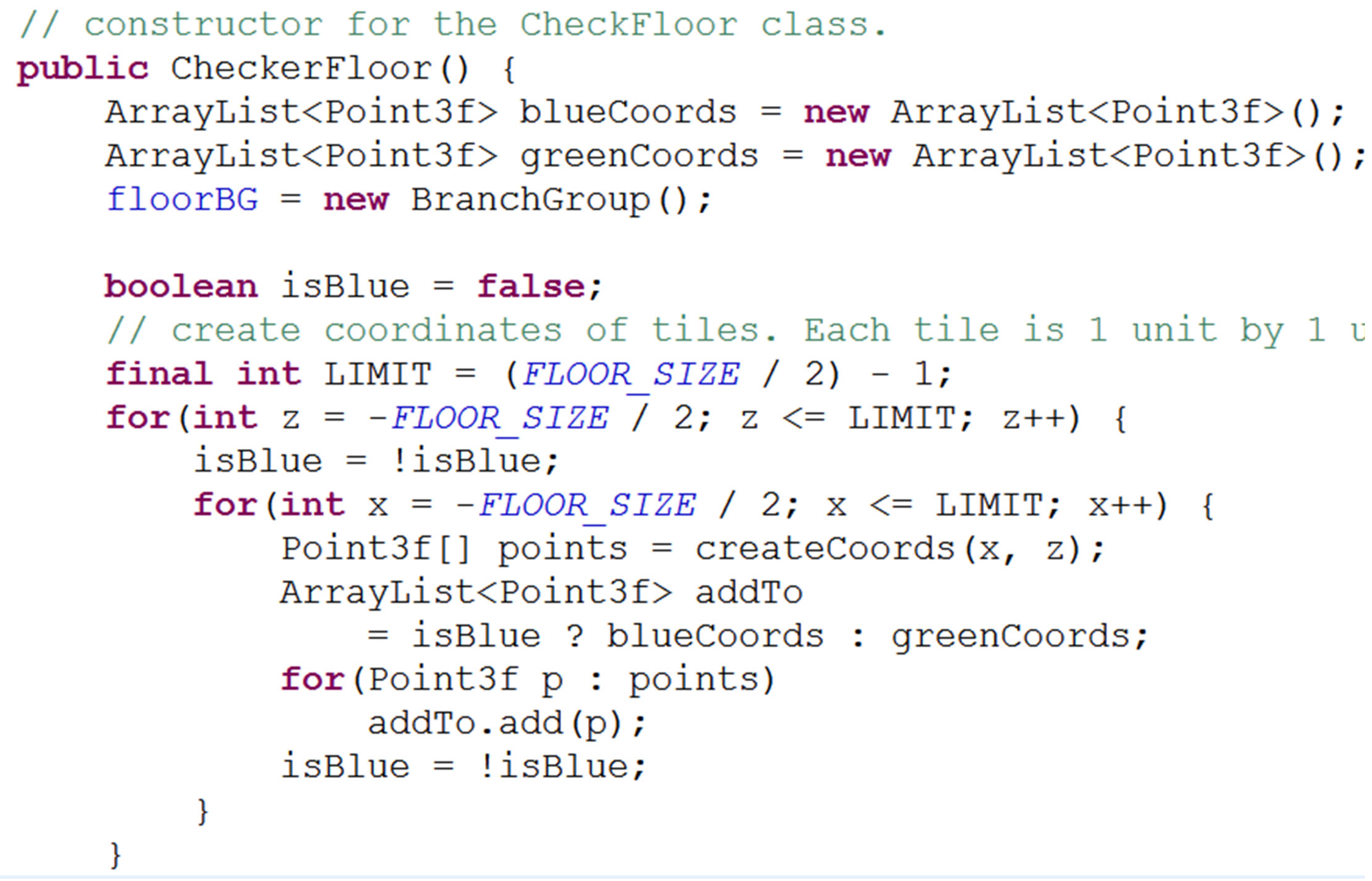




Colored Tiles

- Quad specified with 4 Point3f objects
  - four corners of the quad
  - order significant
  - "front" of the shape is counterclockwise loop formed by the points
- Individual quads do not have to be adjacent to other quads

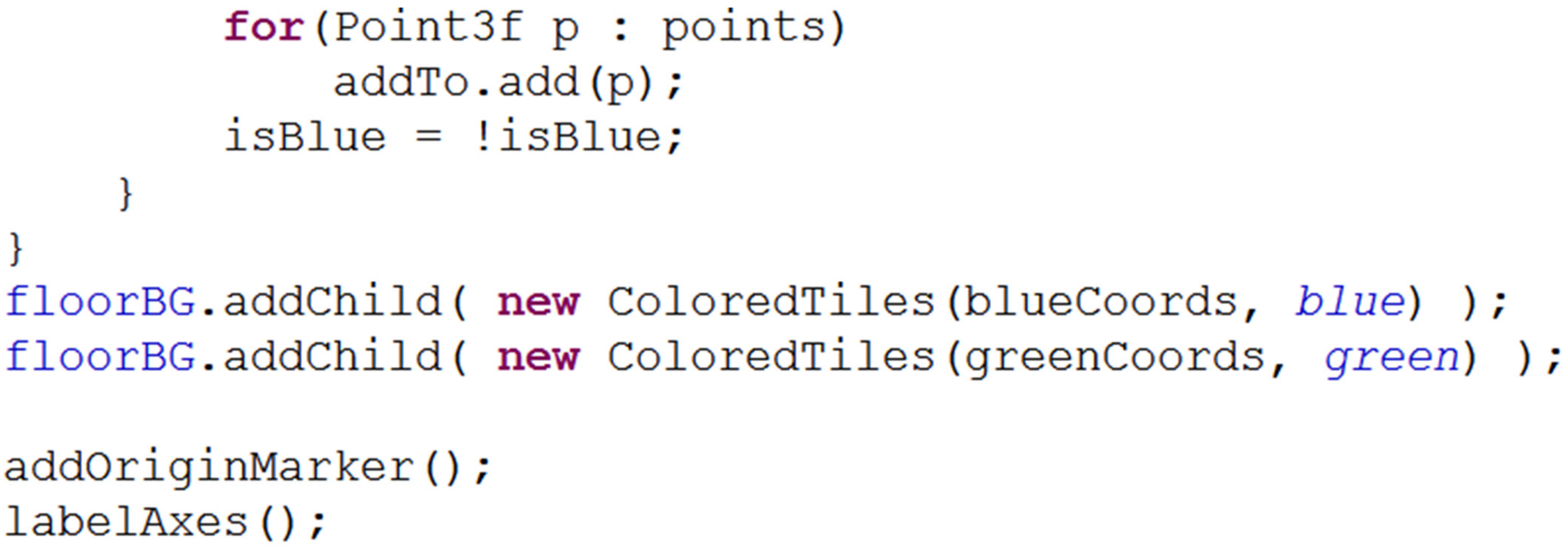
CheckerFloor Constructor



```
// constructor for the CheckFloor class.
public CheckerFloor() {
    ArrayList<Point3f> blueCoords = new ArrayList<Point3f>();
    ArrayList<Point3f> greenCoords = new ArrayList<Point3f>();
    floorBG = new BranchGroup();

    boolean isBlue = false;
    // create coordinates of tiles. Each tile is 1 unit by 1 u
    final int LIMIT = (FLOOR_SIZE / 2) - 1;
    for(int z = -FLOOR_SIZE / 2; z <= LIMIT; z++) {
        isBlue = !isBlue;
        for(int x = -FLOOR_SIZE / 2; x <= LIMIT; x++) {
            Point3f[] points = createCoords(x, z);
            ArrayList<Point3f> addTo
                = isBlue ? blueCoords : greenCoords;
            for(Point3f p : points)
                addTo.add(p);
            isBlue = !isBlue;
        }
    }
```

CheckerFloor Constructor

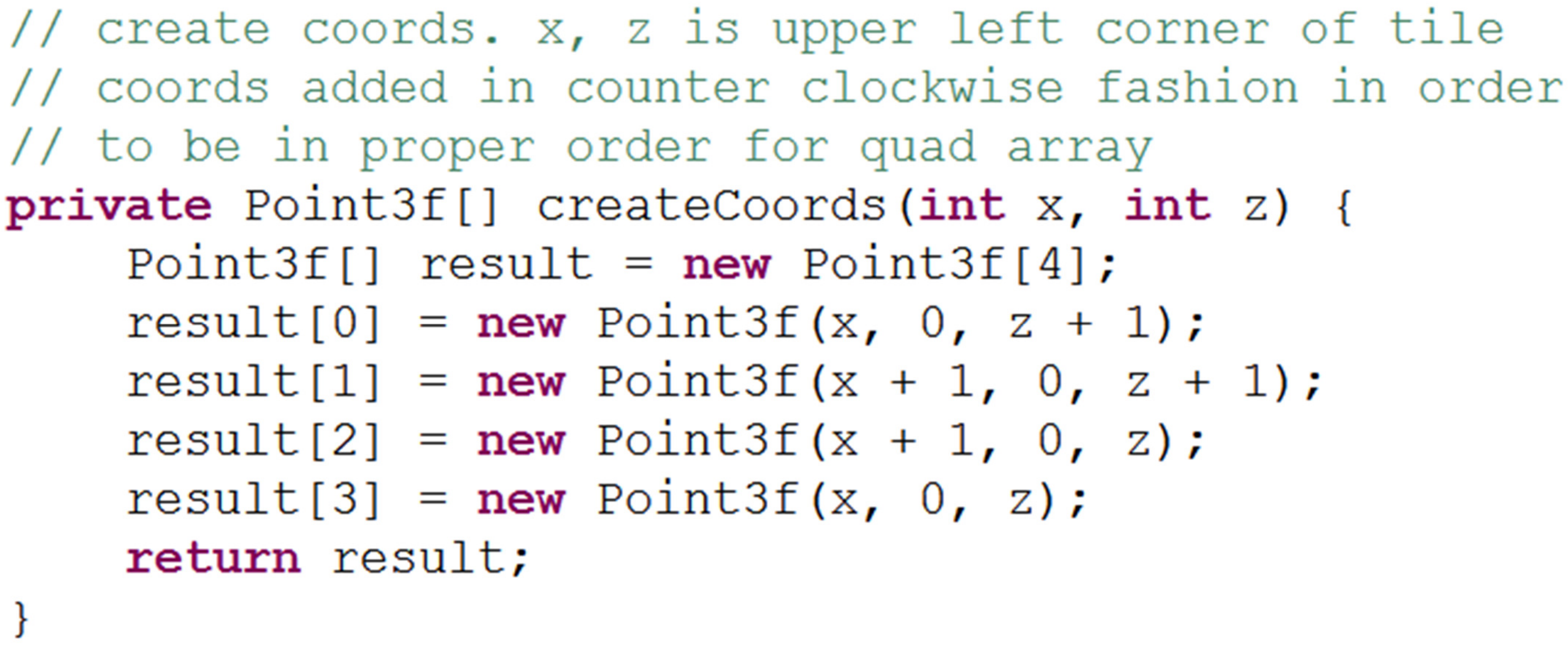
- continued



```
        for(Point3f p : points)
            addTo.add(p);
        isBlue = !isBlue;
    }
}
floorBG.addChild( new ColoredTiles(blueCoords, blue) );
floorBG.addChild( new ColoredTiles(greenCoords, green) );

addOriginMarker();
labelAxes();
```

Creating Points

- Called by constructor to create 4 points for one time based on anchor point

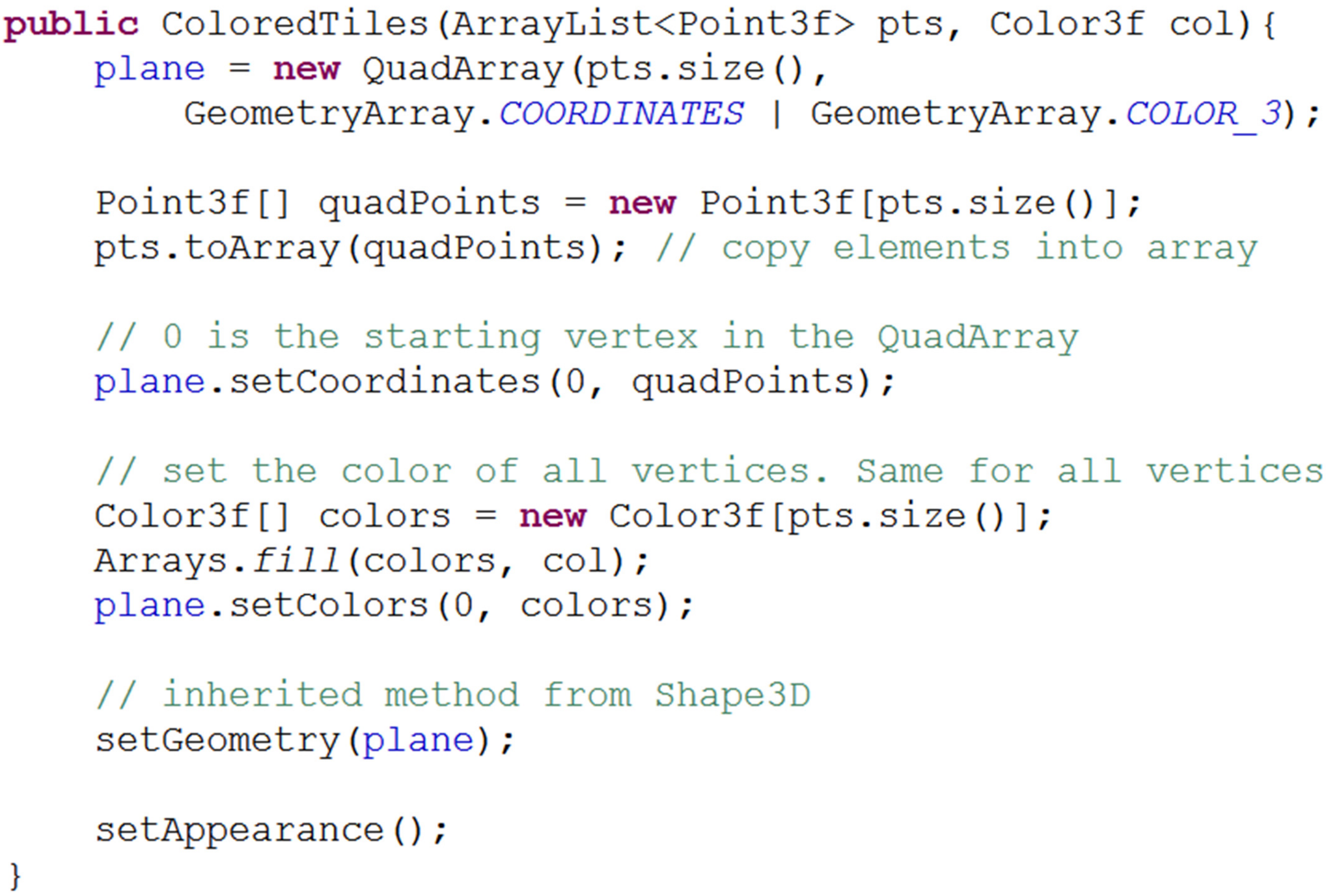


```
// create coords. x, z is upper left corner of tile
// coords added in counter clockwise fashion in order
// to be in proper order for quad array
private Point3f[] createCoords(int x, int z) {
    Point3f[] result = new Point3f[4];
    result[0] = new Point3f(x, 0, z + 1);
    result[1] = new Point3f(x + 1, 0, z + 1);
    result[2] = new Point3f(x + 1, 0, z);
    result[3] = new Point3f(x, 0, z);
    return result;
}
```

colored Tiles

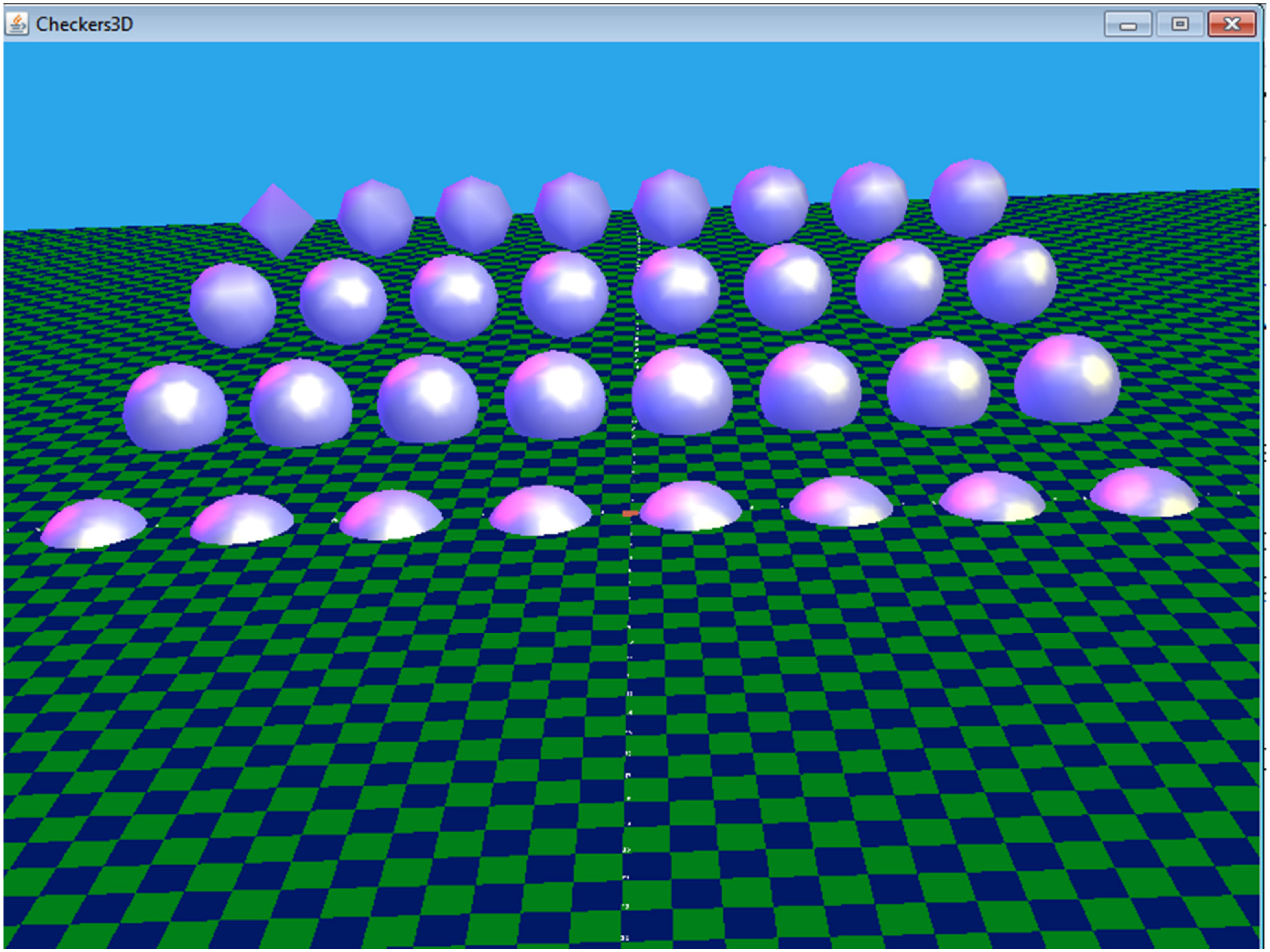
- ArrayList contains 4x Point3f to build QuadArray
- Each tile given a color instead of a material
  - does not react with light
  - must calculate normals if wish to base Appearance on a material

colored Tiles Constructor

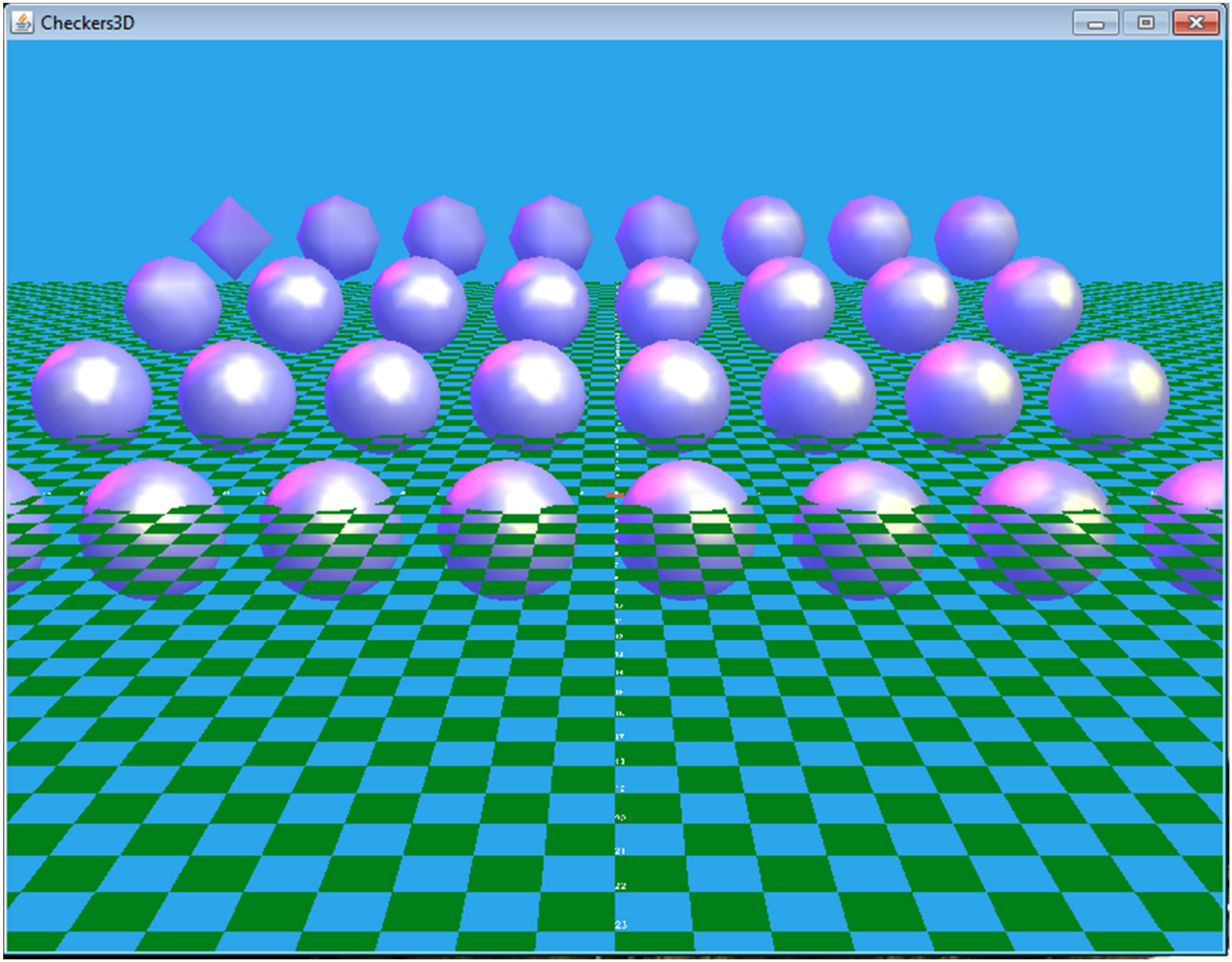


```
public ColoredTiles(ArrayList<Point3f> pts, Color3f col){
    plane = new QuadArray(pts.size(),
        GeometryArray.COORDINATES | GeometryArray.COLOR_3);

    Point3f[] quadPoints = new Point3f[pts.size()];
    pts.toArray(quadPoints); // copy elements into array

    // 0 is the starting vertex in the QuadArray
    plane.setCoordinates(0, quadPoints);

    // set the color of all vertices. Same for all vertices
    Color3f[] colors = new Color3f[pts.size()];
    Arrays.fill(colors, col);
    plane.setColors(0, colors);

    // inherited method from Shape3D
    setGeometry(plane);

    setAppearance();
}
```

Checker Floor

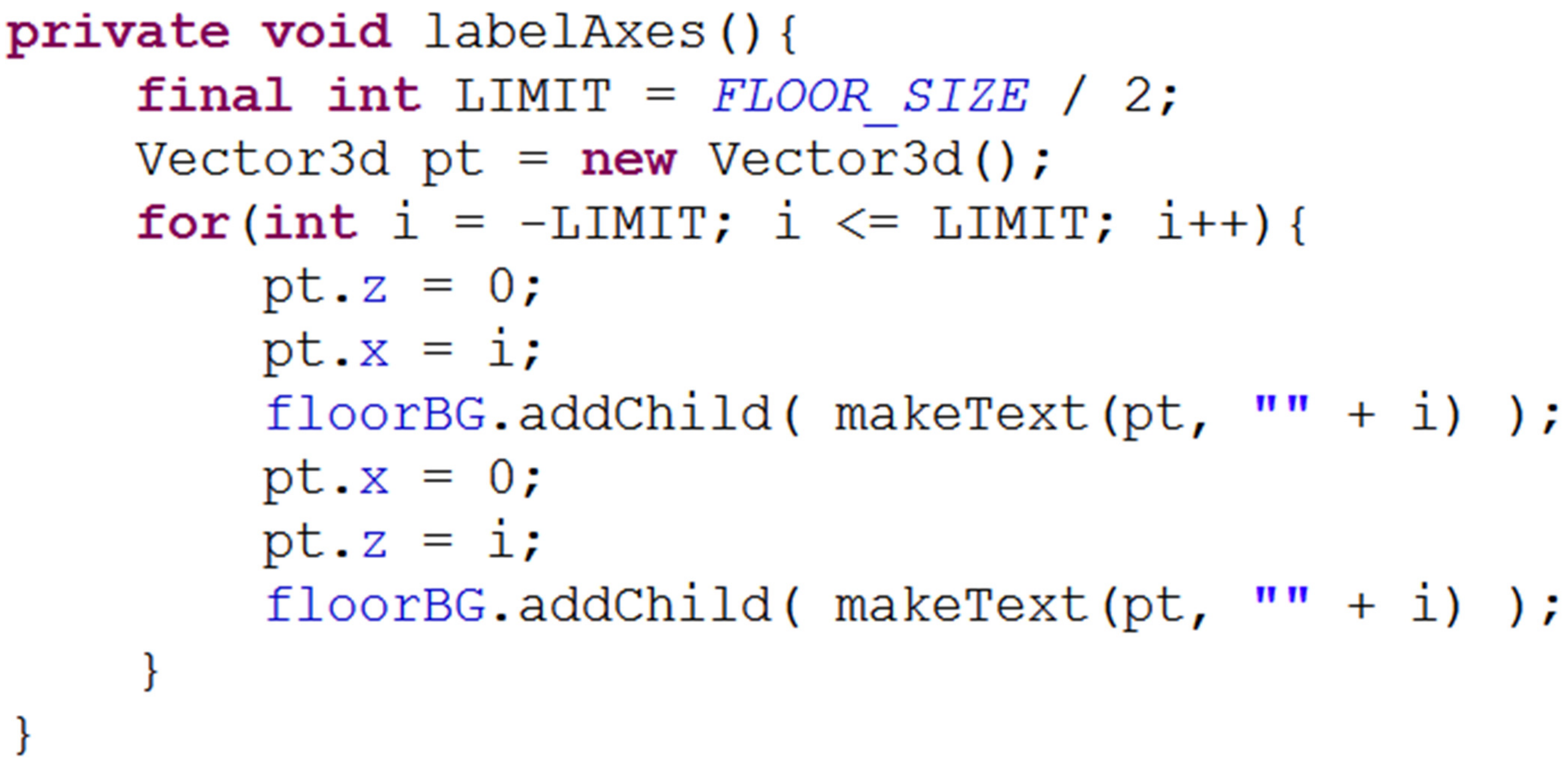


Checker floor - Only Green Tiles



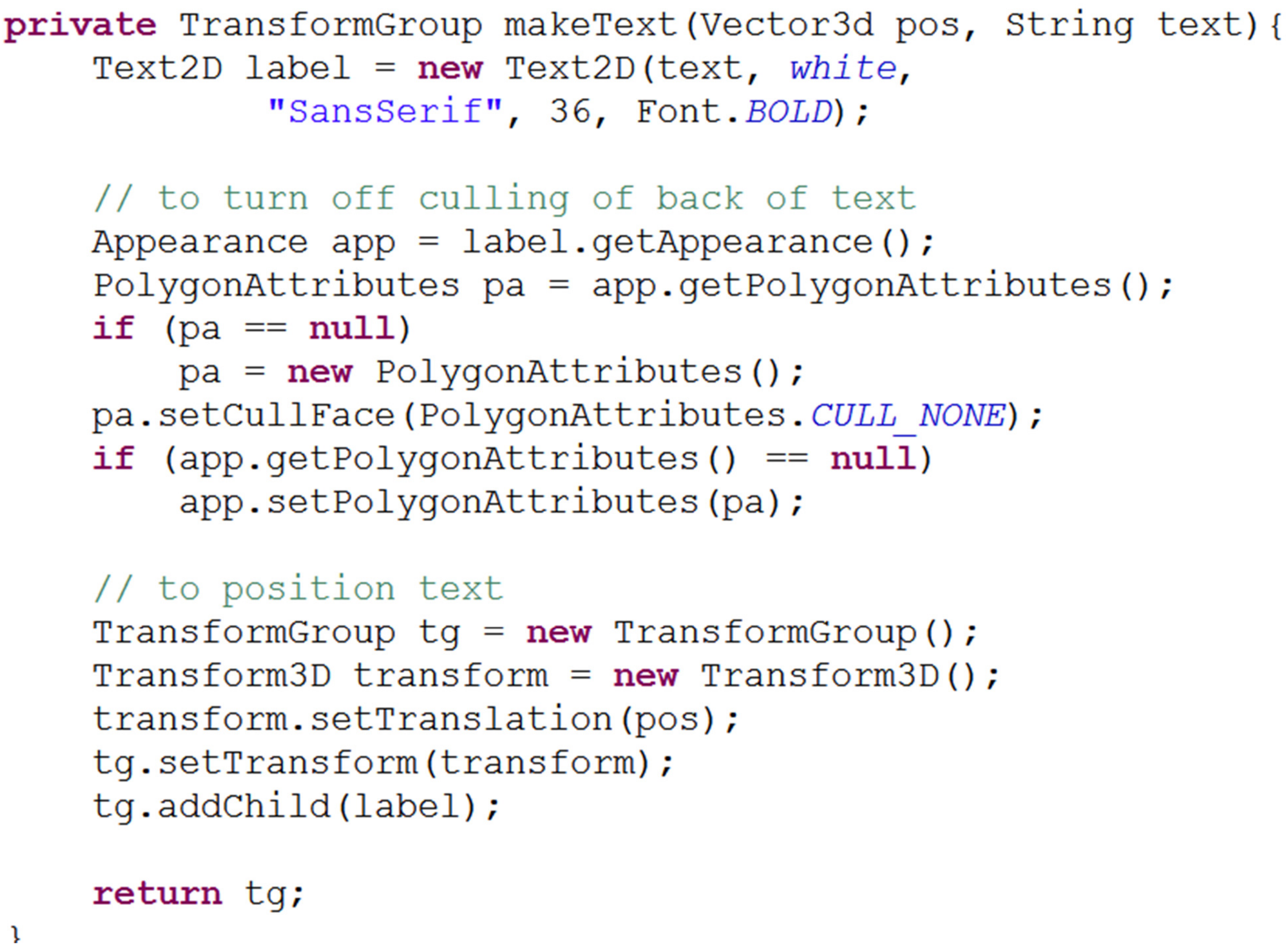
Adding Labels

- Back in CheckerFloor class
  - two methods



```
private void labelAxes(){
    final int LIMIT = FLOOR_SIZE / 2;
    Vector3d pt = new Vector3d();
    for(int i = -LIMIT; i <= LIMIT; i++){
        pt.z = 0;
        pt.x = i;
        floorBG.addChild( makeText(pt, "" + i) );
        pt.x = 0;
        pt.z = i;
        floorBG.addChild( makeText(pt, "" + i) );
    }
}
```

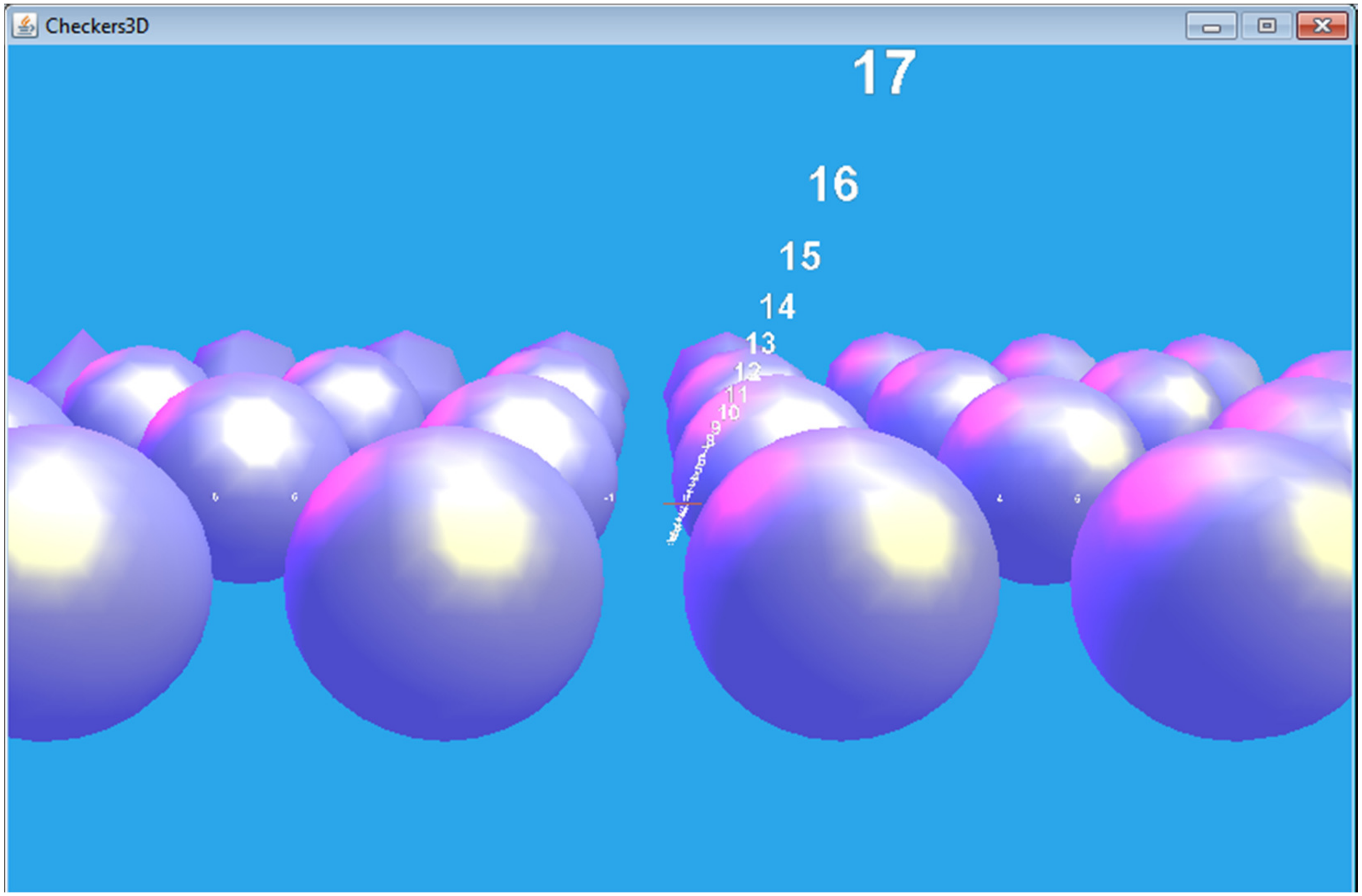
Adding Labels



```java
private TransformGroup makeText(Vector3d pos, String text){
    Text2D label = new Text2D(text, white,
            "SansSerif", 36, Font.BOLD);

    // to turn off culling of back of text
    Appearance app = label.getAppearance();
    PolygonAttributes pa = app.getPolygonAttributes();
    if (pa == null)
        pa = new PolygonAttributes();
    pa.setCullFace(PolygonAttributes.CULL_NONE);
    if (app.getPolygonAttributes() == null)
        app.setPolygonAttributes(pa);

    // to position text
    TransformGroup tg = new TransformGroup();
    Transform3D transform = new Transform3D();
    transform.setTranslation(pos);
    tg.setTransform(transform);
    tg.addChild(label);

    return tg;
}
```

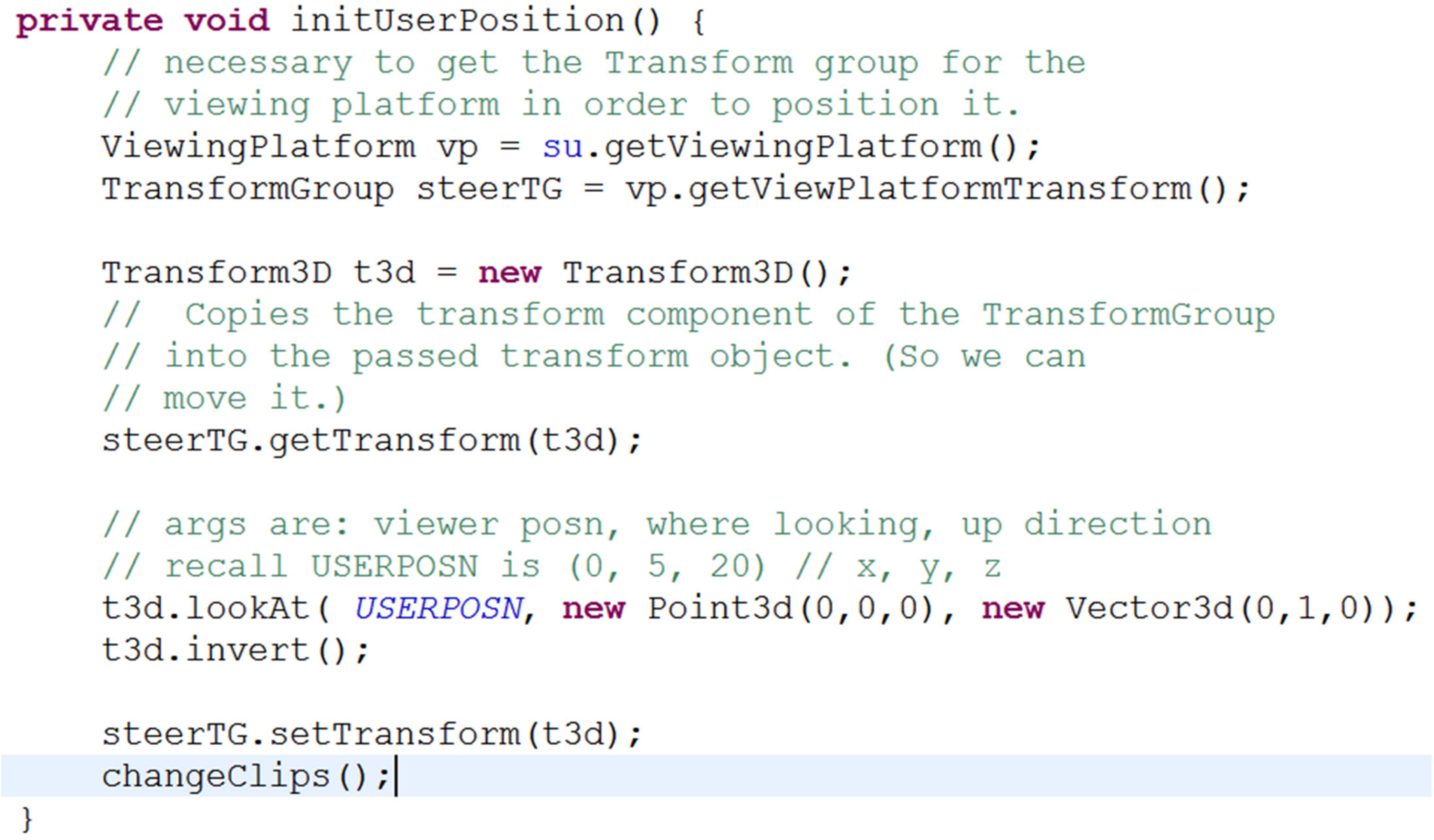
Effects of Culling

- demo program when culling performed on colored Tiles and text



Creating and Positioning Camera

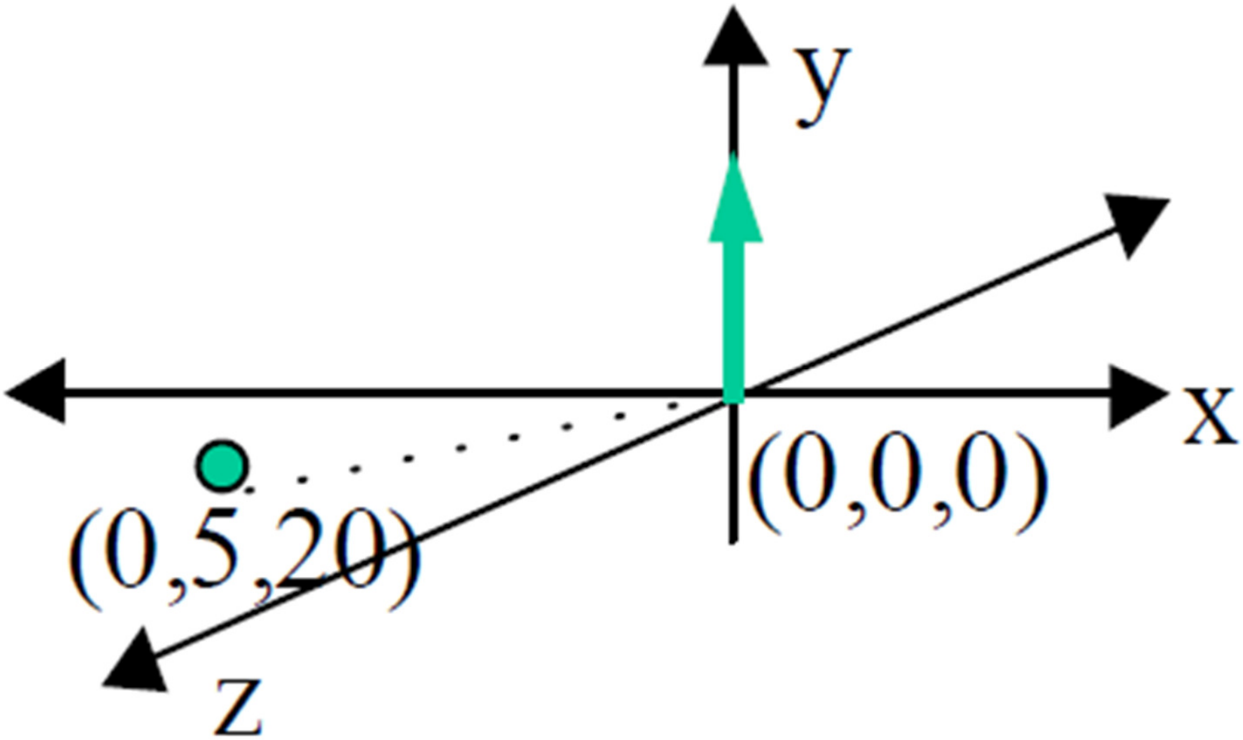
- Back in WrapCheckers3D class



```
private void initUserPosition() {
    // necessary to get the Transform group for the
    // viewing platform in order to position it.
    ViewingPlatform vp = su.getViewingPlatform();
    TransformGroup steerTG = vp.getViewPlatformTransform();

    Transform3D t3d = new Transform3D();
    //  Copies the transform component of the TransformGroup
    // into the passed transform object. (So we can
    // move it.)
    steerTG.getTransform(t3d);

    // args are: viewer posn, where looking, up direction
    // recall USERPOSN is (0, 5, 20) // x, y, z
    t3d.lookAt( USERPOSN, new Point3d(0,0,0), new Vector3d(0,1,0));
    t3d.invert();

    steerTG.setTransform(t3d);
    changeClips();
}
```

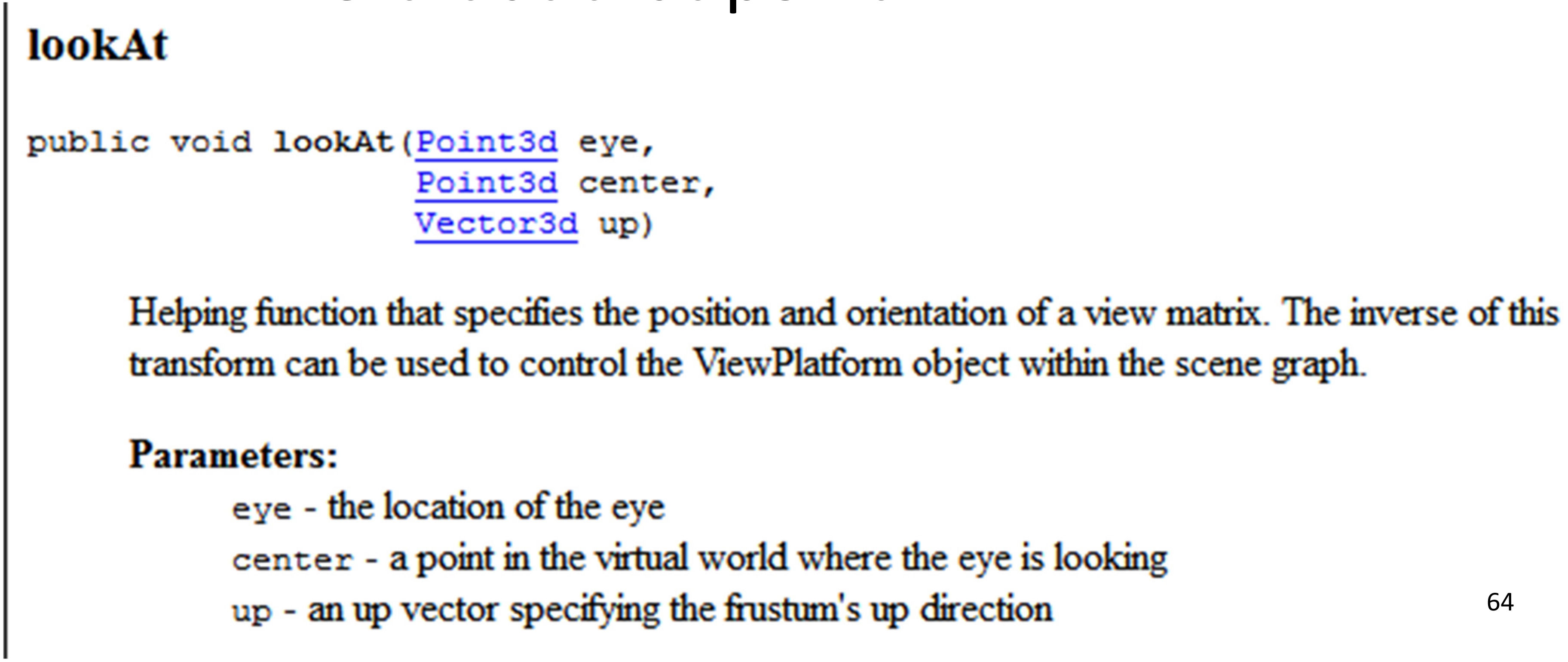
Initial Position

- SimpleUniverse create viewing platform for us
- lookAt method to set position



lookAt method

- the lookAt method makes the object being translated face towards the ViewPlatform, which actually makes the ViewPlatform face exactly away from our scene, so we invert it at that point.



lookAt

```
public void lookAt(Point3d eye,
                   Point3d center,
                   Vector3d up)
```

Helping function that specifies the position and orientation of a view matrix. The inverse of this transform can be used to control the ViewPlatform object within the scene graph.

Parameters:

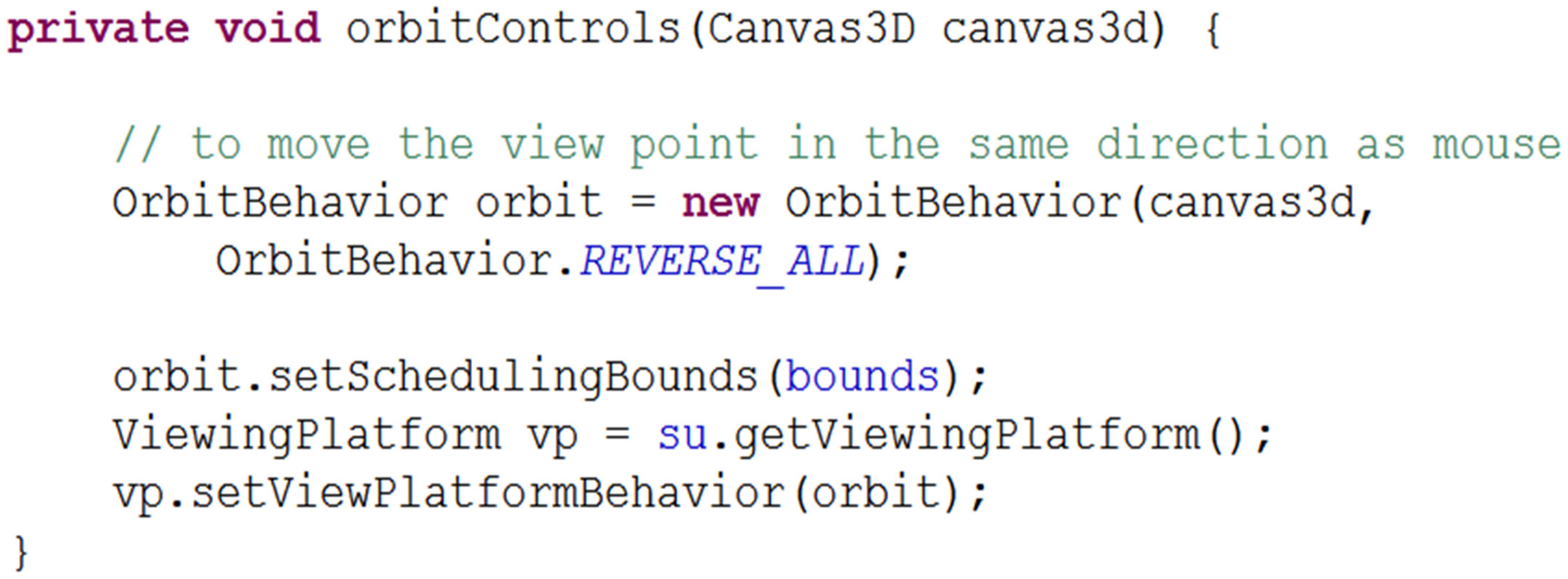
`eye` - the location of the eye

`center` - a point in the virtual world where the eye is looking

`up` - an up vector specifying the frustum's up direction

Orbit Controls

- Simple way to allow mouse to move the viewing platform



```java
private void orbitControls(Canvas3D canvas3d) {

    // to move the view point in the same direction as mouse
    OrbitBehavior orbit = new OrbitBehavior(canvas3d,
        OrbitBehavior.REVERSE_ALL);

    orbit.setSchedulingBounds(bounds);
    ViewingPlatform vp = su.getViewingPlatform();
    vp.setViewPlatformBehavior(orbit);
}
```