JOGL Live Rendering Techniques in Data Acquisition Systems

Yannick Le Goc, Institut Laue-Langevin

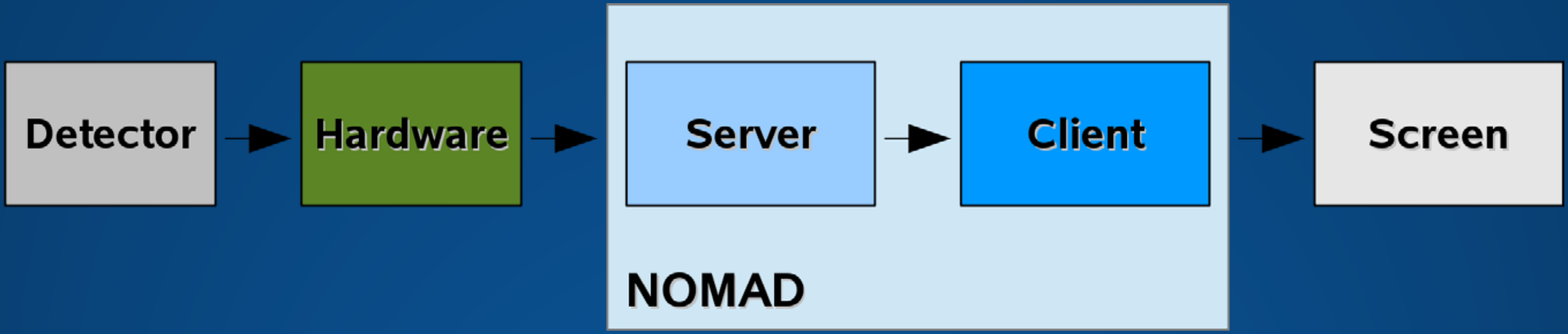
ICALEPCS 2013

11 october 2013

Outline

- Data Acquisition Chain
- JOGL Choice
- Draw Detector Data
- 3 Techniques in JOGL

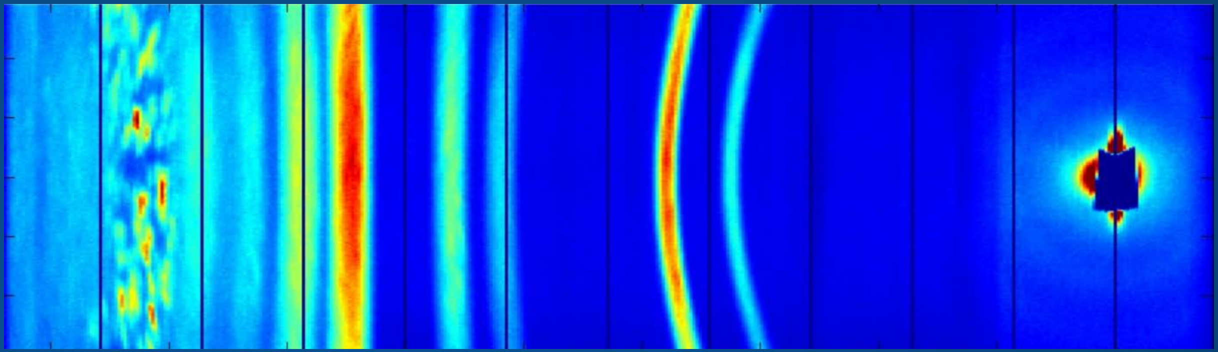
Data Acquisition Chain



- NOMAD
  - C++ Server
  - Java SWT Client

Data Acquisition Chain

- Different detector geometries and sizes
  - Can be small, 1 pixel
  - Can be big, 4K x 4K pixels



(IN5 detector image)

- Different acquisition frequencies
  - From 0.01Hz to 5MHz

Data Acquisition Chain

- Plot requested refresh frequency : 10Hz

- How to visualize such a large quantity of data at high frequency?

Data Acquisition Chain

- Plot requested refresh frequency : 10Hz

- How to visualize such a large quantity of data at high frequency?

Need for a performant live rendering

Existing Libraries

- Python Library
  - GuiQWT
  - PyQtGraph

⇨ Too difficult to integrate

- Java Library
  - TANGO
  - Jzy3d

⇨ Easy to integrate but not performant enough

Solution

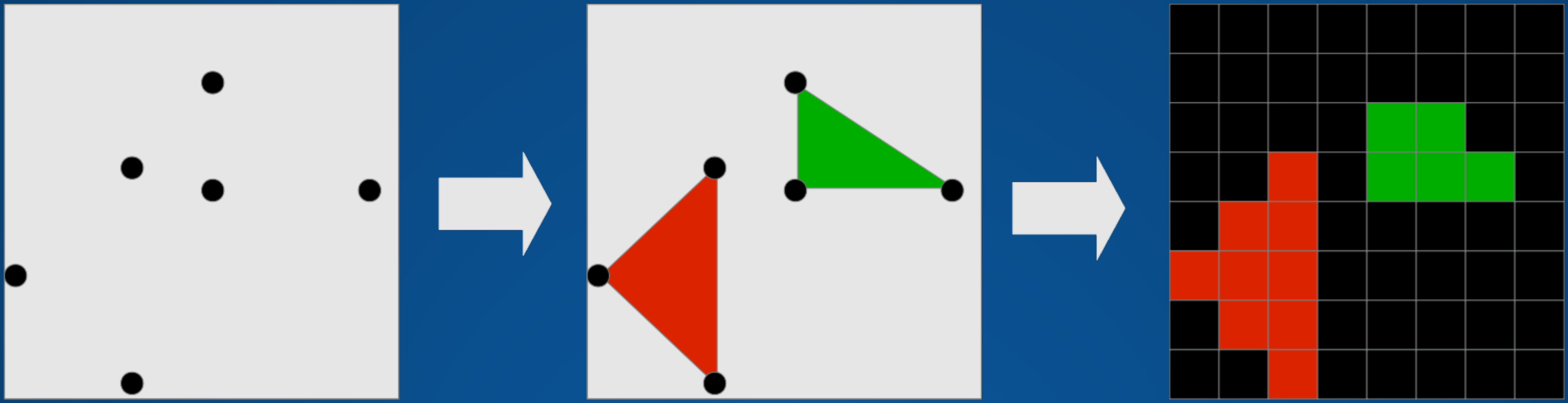
- JOGL
  - OpenGL binding in Java
  - Close to the graphics card
  - Optimized rendering guarantee
  - Current version 2.0.2 supports OpenGL 4.3

What is OpenGL?

- API for interacting with the GPU
- State machine
- Very simplified pipeline

What is OpenGL?

- API for interacting with the GPU
- State machine
- Very simplified pipeline

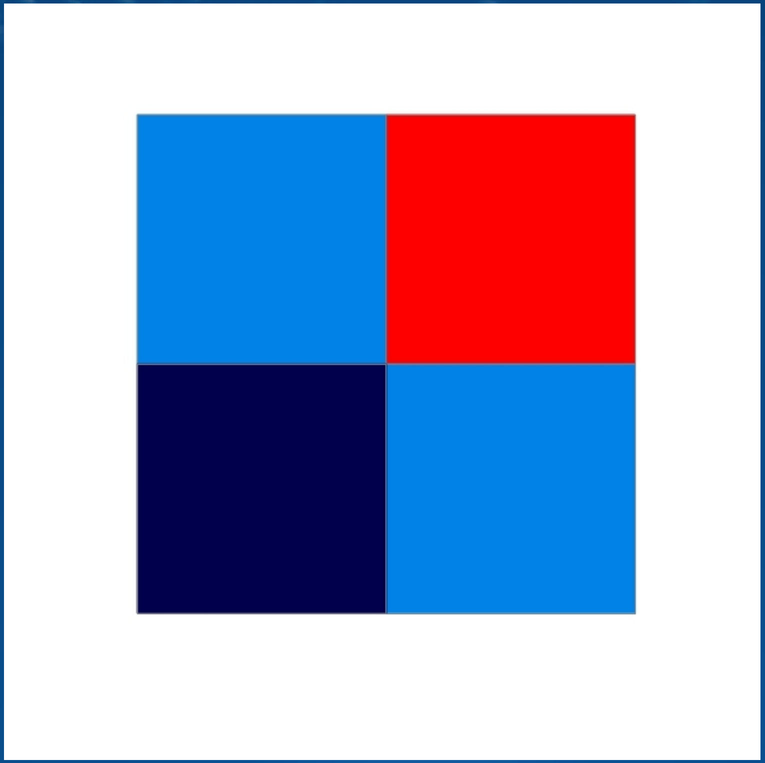


Primitive
Assembly

Projection
Rasterization

Draw Detector Data

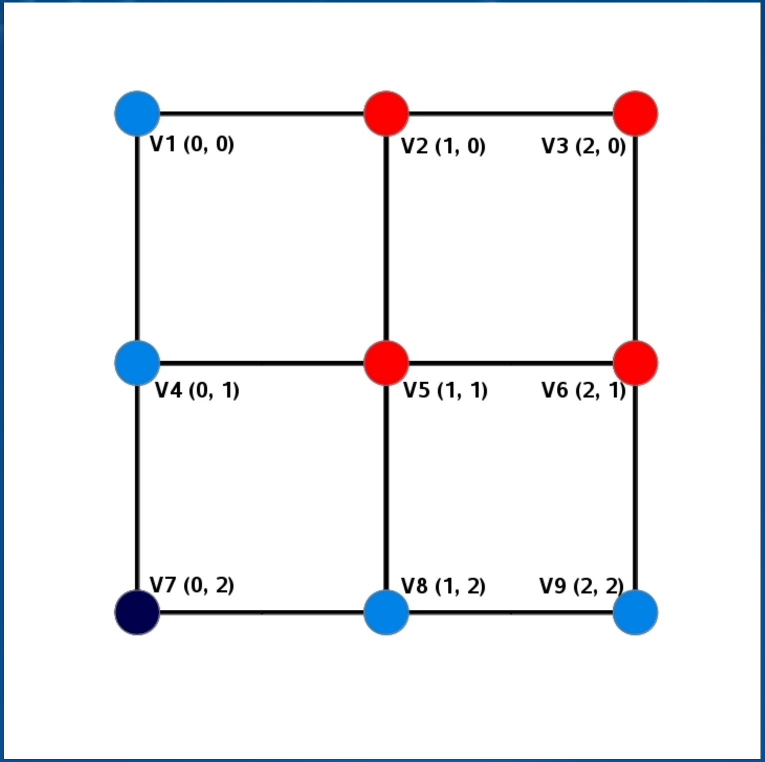
- 2D detector data visualized as an array of pixels



- How to convert pixels into vertices?

Draw Detector Data

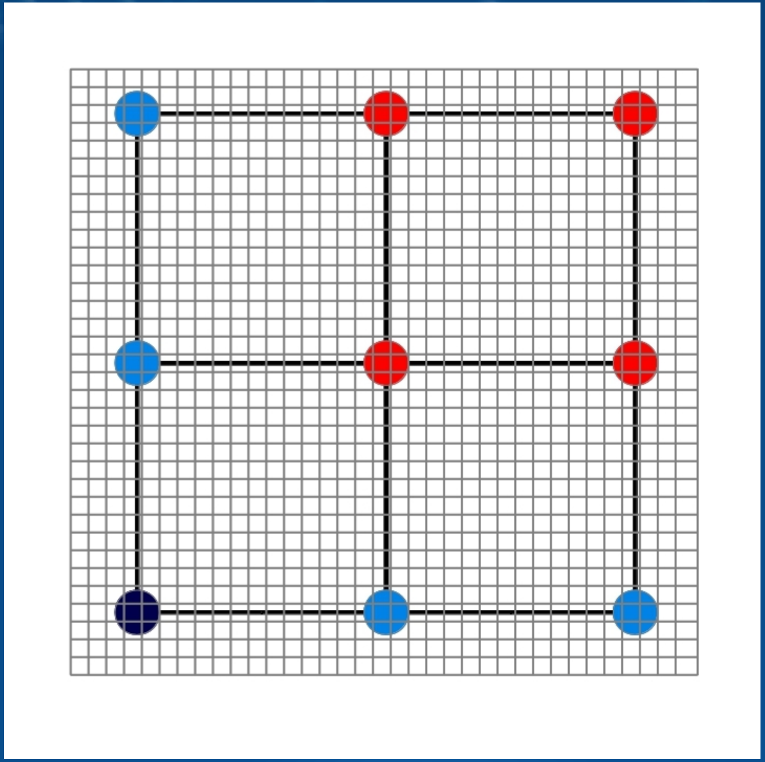
- Detector data transformed into vertices



- Vertices are shared!

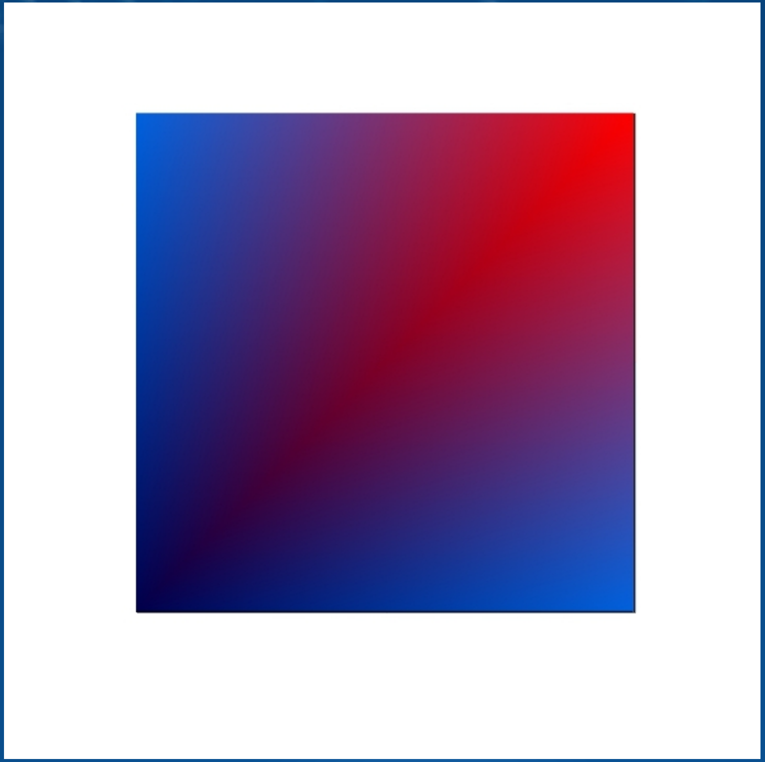
Draw Detector Data

- Rasterization



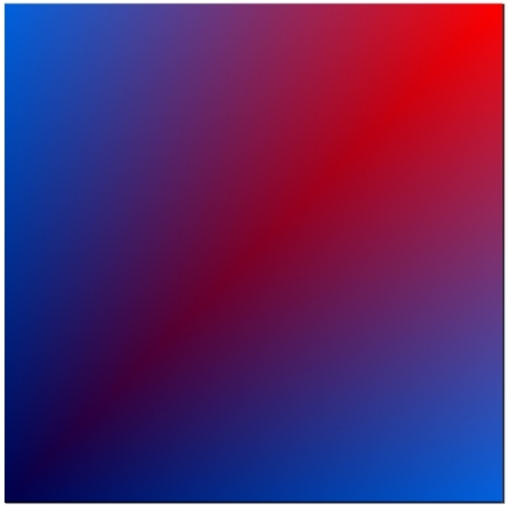
Draw Detector Data

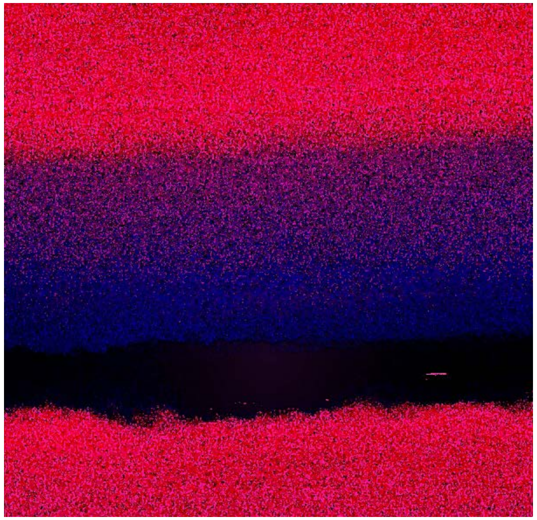
- Smooth rendering



Draw Detector Data

- Smooth rendering



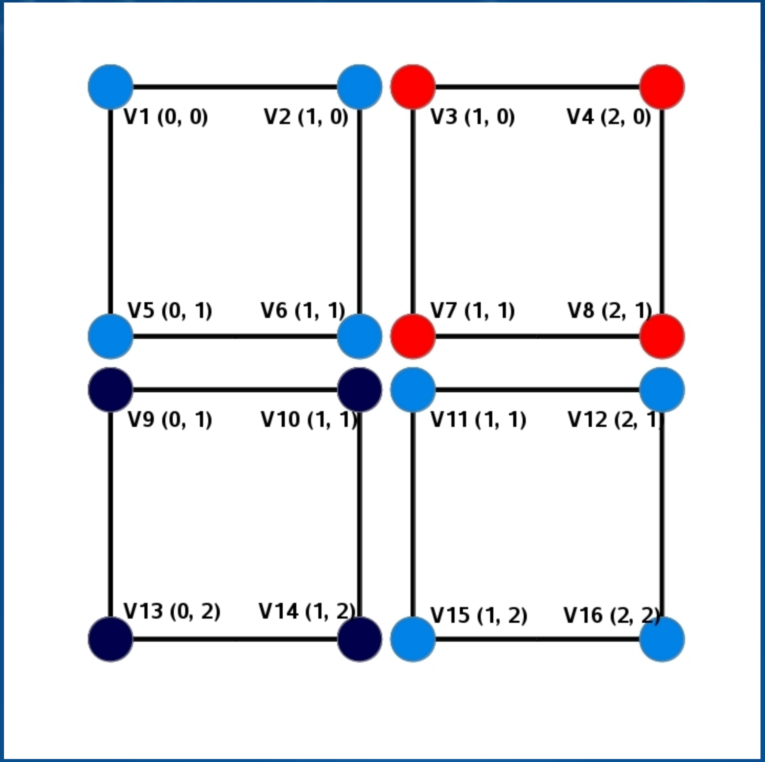


Rothko II

- Not the visualization we want !

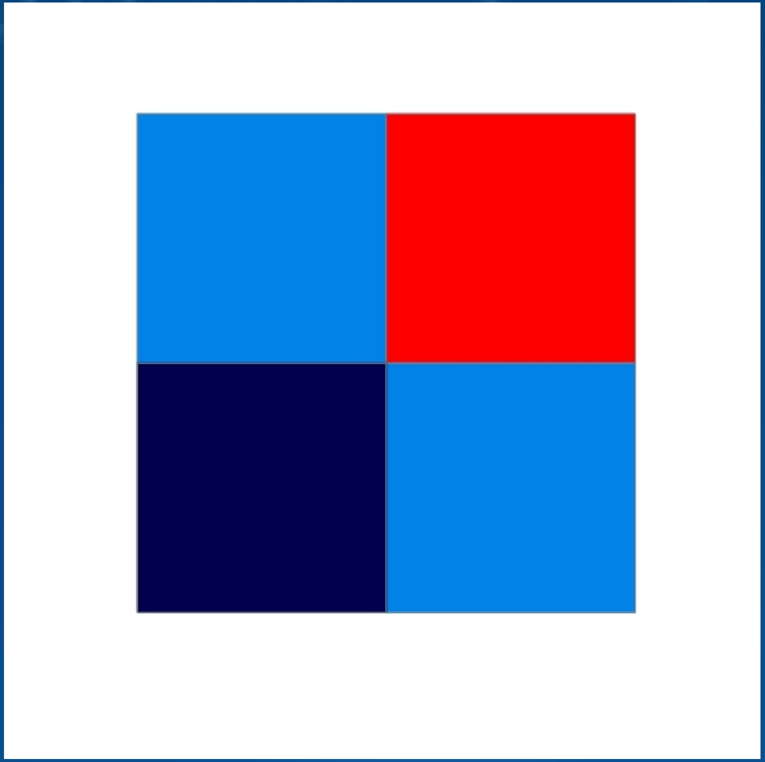
Draw Detector Data

- Quadruple the vertices



Draw Detector Data

- Non-smooth rendering



Code with JOGL

- Technique 1: *Immediate Mode*

```
void display(GL gl) {

        gl.glBegin(GL.GL_QUADS);
        ...
        gl.glColor3f(r1, g1, b1);
        gl.glVertex2f(v1.x, v1.y);
        gl.glVertex2f(v2.x, v2.y);
        gl.glVertex2f(v6.x, v6.y);
        gl.glVertex2f(v5.x, v5.y);
}
```

- Simple, but too many calls to OpenGL
- More than 16K calls for 4K x 4K detectors !

Code with JOGL

- Technique 2: *Vertex Arrays*

```
void display(GL gl) {

        fillBuffers();
        drawBuffers(gl);
}
```

Code with JOGL

- Technique 2: *Vertex Arrays*

```
void fillBuffers() {

        ...
        vertexBuffer.put(v1.x);
        vertexBuffer.put(v1.y);
        vertexBuffer.put(v2.x);
        vertexBuffer.put(v2.y);
        ...
        colorBuffer.put(r1);
        colorBuffer.put(g1);
        colorBuffer.put(b1);
        ...
}
```

Code with JOGL

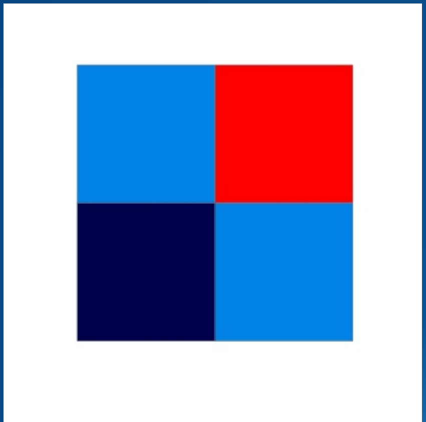
- Technique 2: *Vertex Arrays*

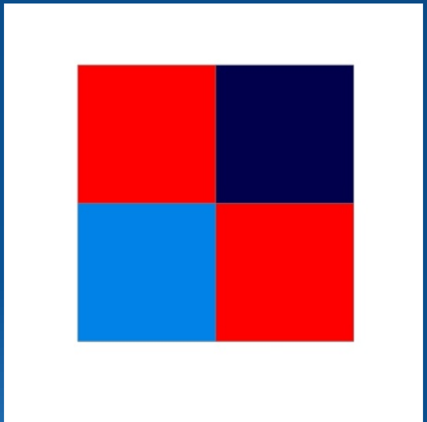
```
void drawBuffers(GL gl) {

        gl.glVertexPointer(2, GL.GL_FLOAT, 0,
                                        vertexBuffer);
        gl.glColorPointer(3, GL.GL_UNSIGNED_BYTE, 0,
                                        colorBuffer);
        gl.glDrawElements(GL.GL_QUADS, size,
                GL.GL_UNSIGNED_INT, indexBuffer);

}
```

- Much better, only 3 OpenGL calls

Code with JOGL

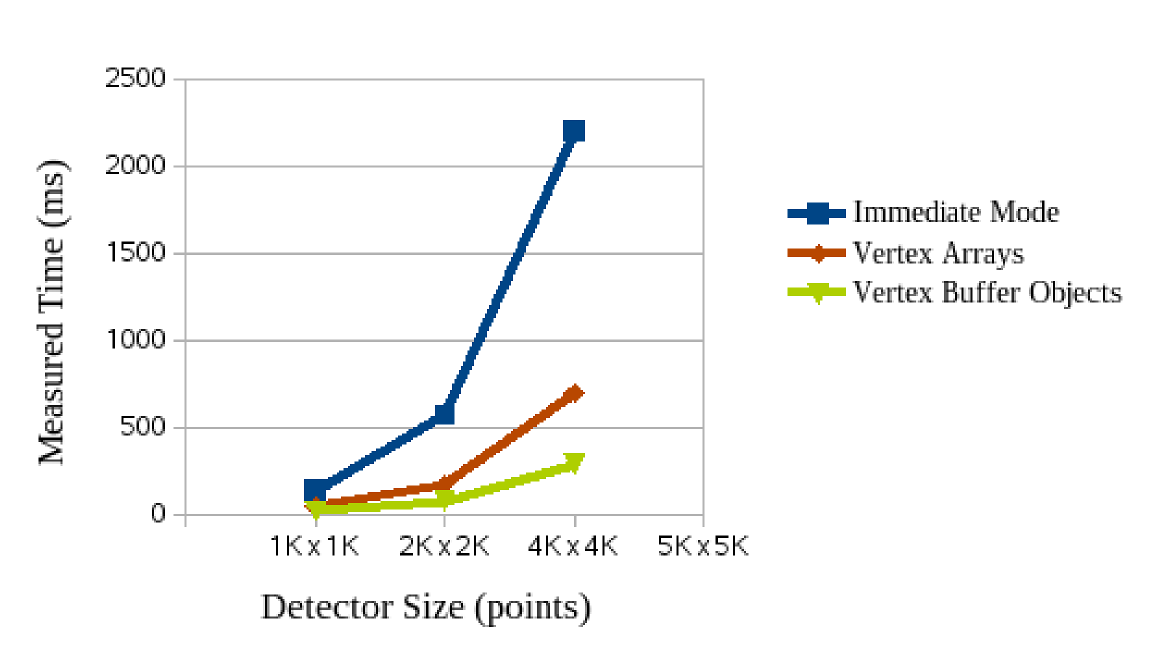
- Technique 3: *Vertex Buffer Objects (VBO)*
  - Keep the vertex buffer in the memory of the GPU
  - Only transfer the color buffer





JOGL Techniques Comparison

- Drawing times



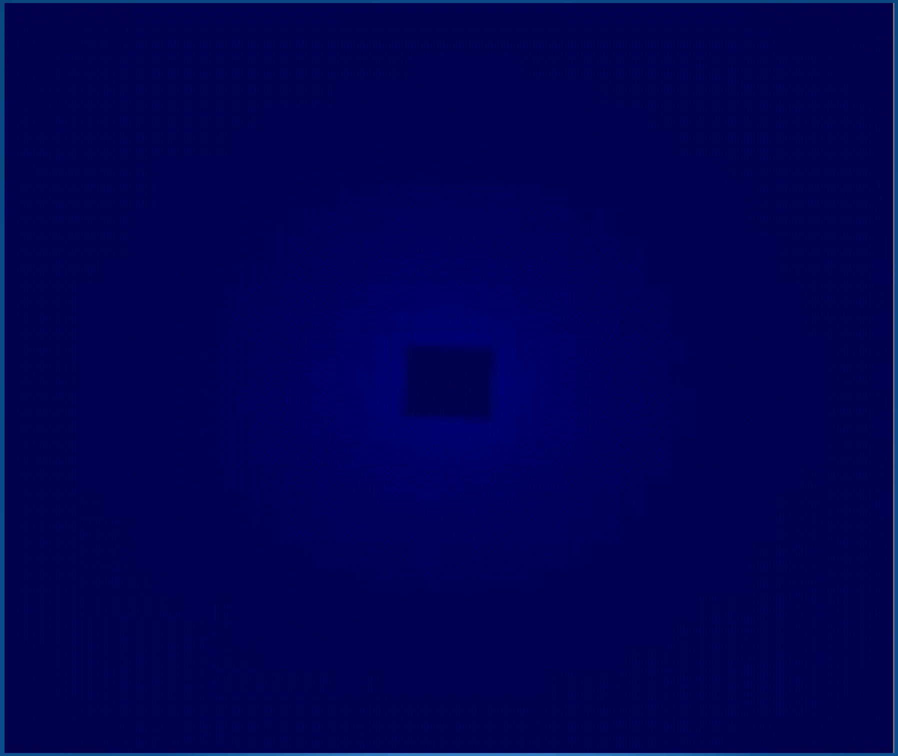
- *VBO* 10 times faster than *Immediate Mode*

Conclusion

- Advanced technique with VBO
- Very efficient rendering with JOGL
- Satisfies instrument requirements

Conclusion

- Advanced technique with VBO
- Very efficient rendering with JOGL
- Satisfies instrument requirements



Thank You

- Any questions?

Contact: legoc@ill.fr