CS3388B: Lecture 6

January 31, 2023

6 Drawing in 2D using Open GL: Part 2

By now we know how to draw lines, polylines, transform them, and clip them. In the 2D world there is still a couple of things to finish:

- Rasterization
- Splines and Curves (to be continued).

6.1 Rasterization

Rasterization is the process of transforming the vector-defined (floating-point-defined) shapes and graphics primitives into a *raster image* that can then be drawn pixel-by-pixel.

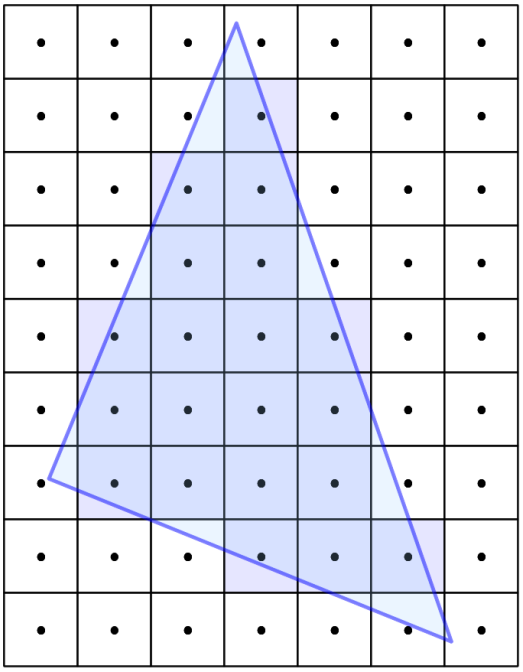
Recall that the viewport transformation maps NDC to the "canvas" on which we will actually draw. The viewport canvas is the subset of the window in which we are drawing. However, even after applying the viewport transformation, all of our primitives are still defined using vertices with floating-point numbers and symbolic formulas (e.g. a line segment between two points is $y - y_0 = \frac{y_1 - y_0}{x_1 - x_0} x - x_0$).

The process of **rasterizing** is the transformation of converting that symbolic and continuous data to a discrete set of pixels. In fact, more than pixels. The process of rasterization produces **fragments**. A fragment is the combination of all the data needed to color a pixel:

1. raster position (pixel coordinates)
2. color
3. texture coordinates
4. depth
5. alpha
6. etc.

There are many different methods and algorithms for rasterization. Different graphics libraries use different algorithms.

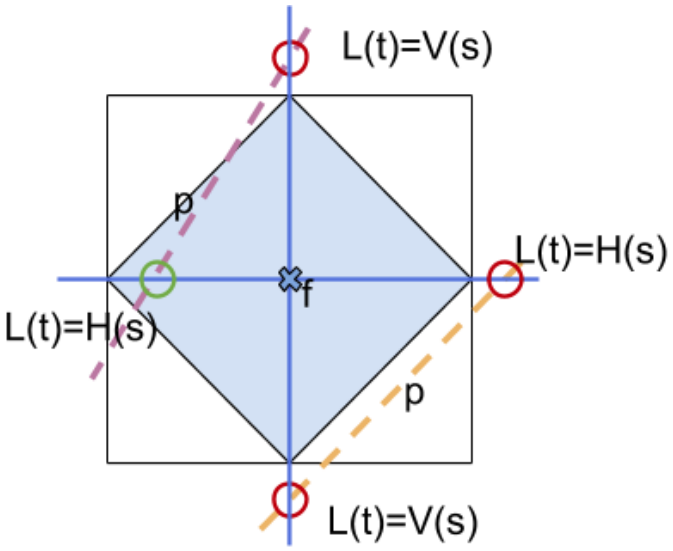
But, they all follow a simple idea: take the viewport and overlay it on a grid of pixels and *test points*, typically the center of each pixel. Then, apply some algorithm to determine whether a particular primitive "covers" a pixel, and then color that pixel appropriately.



6.1.1 Line Rasterization

The classic line rendering algorithm is **Bresenham's line algorithm**. But let's consider something more modern. Let's look at the line rasterization in DirectX 11 and modern OpenGL. What makes this "modern" is that it can create so-called *non-Bresenham fragments*.

This uses a **Diamond exit rule**.



For a pixel centered at (x_f, y_f), define the diamond-shaped region:

$$\left\{(x, y) \;\middle|\; |x - x_f| + |y - y_f| \leq \frac{1}{2}\right\}$$

A line covers a pixel if the line **exits** the pixel's diamond **test area** when travelling along the line from the start towards the end.

What is a test area you ask? The test area (really test edges) depends on the line's slope m.

1. If $|m| \leq 1$, then the test area is the diamond's bottom two edges *and* the bottom corner
2. Otherwise, the test area is the diamond's bottom two edges *and* the bottom corner and the right corner.

If $|m| \leq 1$ the line is called **x-major**. Otherwise it is called **y-major**.

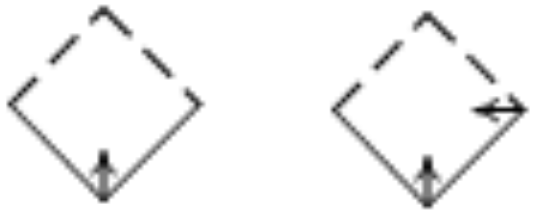
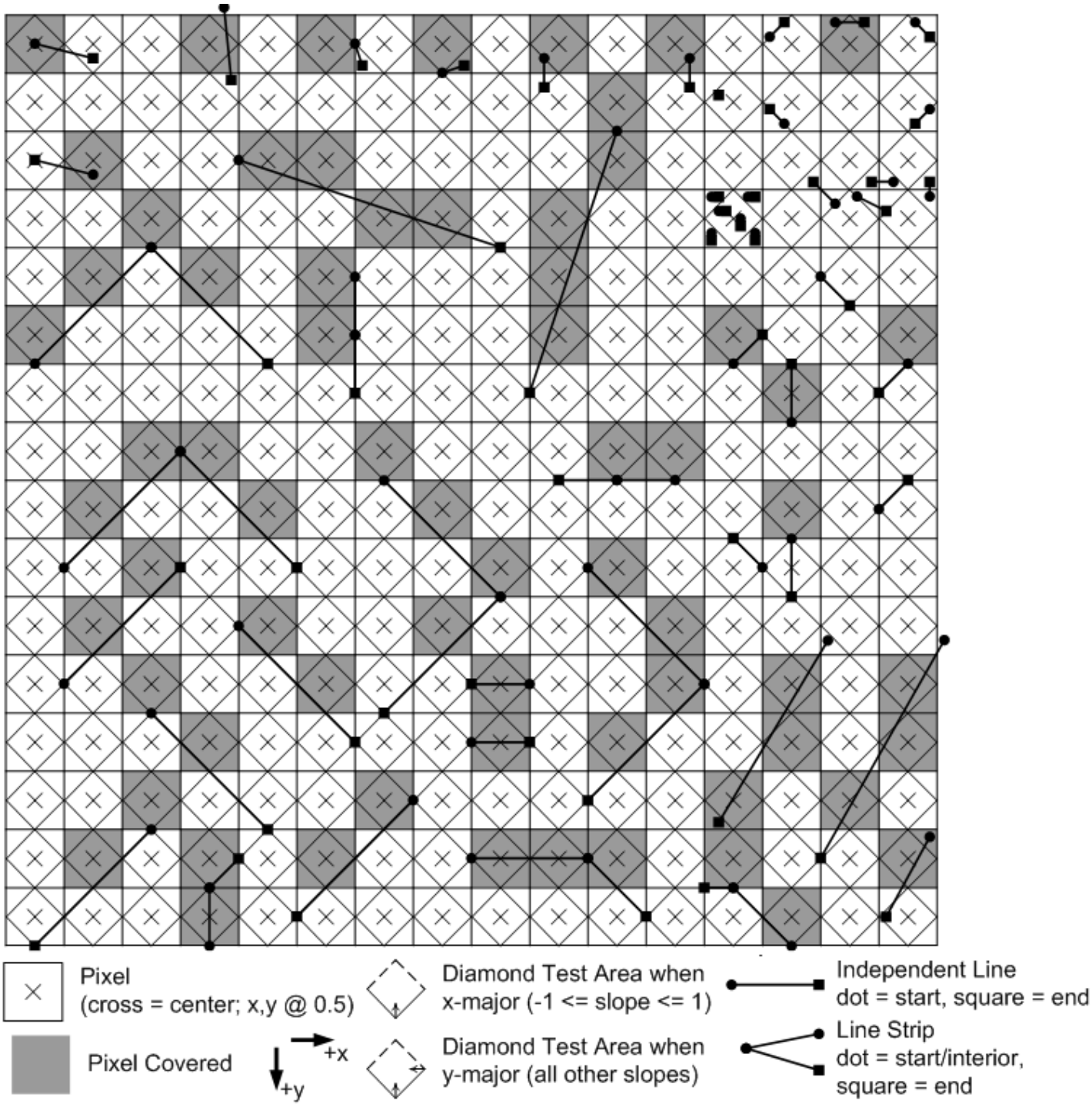


Figure 1: x-major on the left, y-major on the right.

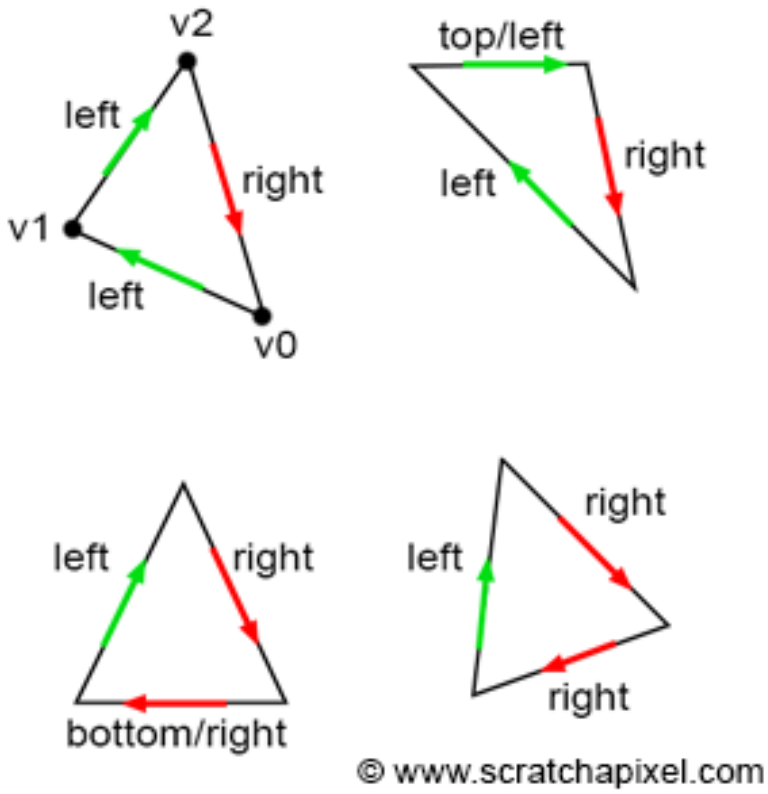
If a line exits the diamond's test area, or is tangent to the test area, the entire pixel gets colored in. Notice that 3 corners in x-major (2 corners in y-major) are excluded from the test area. If a line merely *touches* one of the corners in the test area, then the pixel is colored. Otherwise, it is not.


Pixel
(cross = center; x,y @ 0.5)
Pixel Covered
+x
+y
Diamond Test Area when
x-major (-1 <= slope <= 1)
Diamond Test Area when
y-major (all other slopes)
Independent Line
dot = start, square = end
Line Strip
dot = start/interior,
square = end

6.1.2 Triangle Rasterization

Triangle rasterization is a lot easier than line rasterization. We define a region enclosed by the 3 edges of a triangle. If the pixel's test point falls *inside* this region, color in that pixel.

Revisiting the previous rasterized triangle, we see that the test points are the pixel's center and each pixel which is colored has its test point fall fully within the triangle's region.



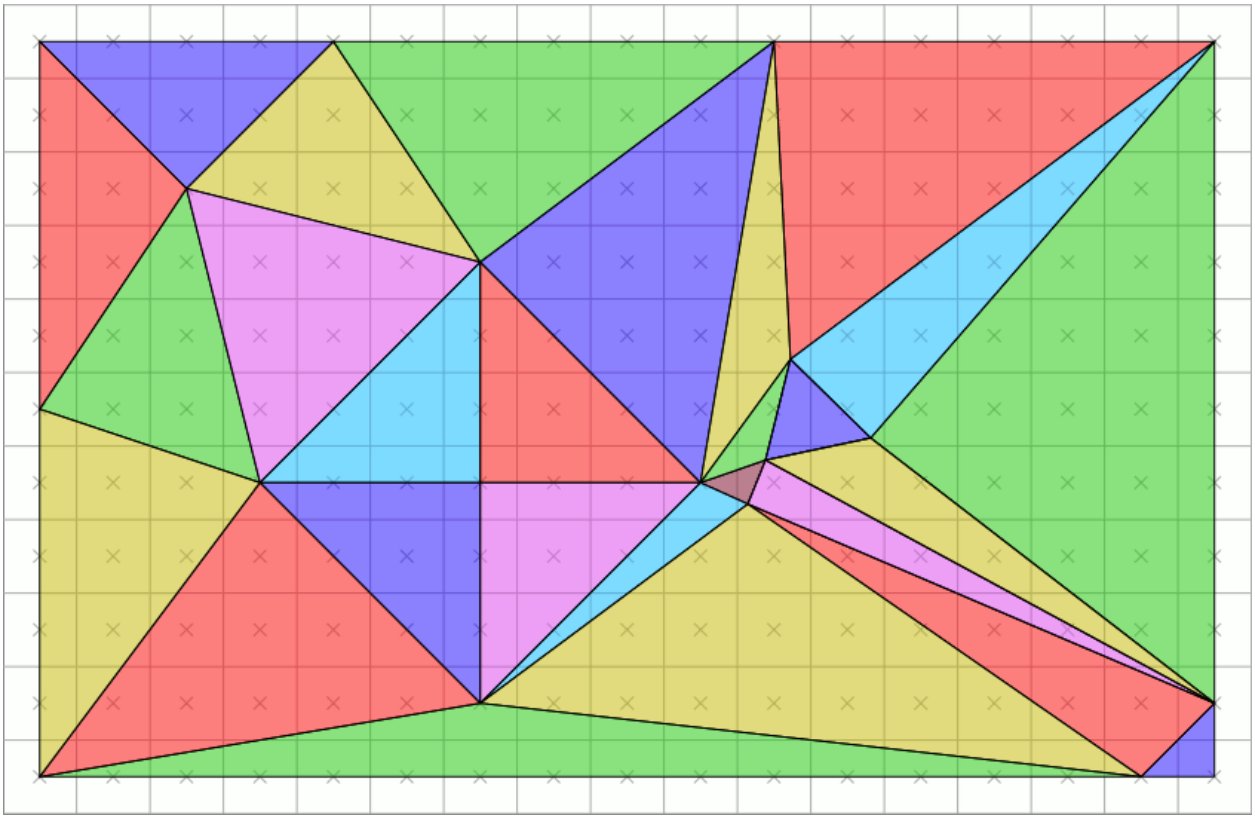
But what if a test point falls *on* a triangle. Not within? In particular, what if two triangles share and edge and that edge goes directly through the test point?

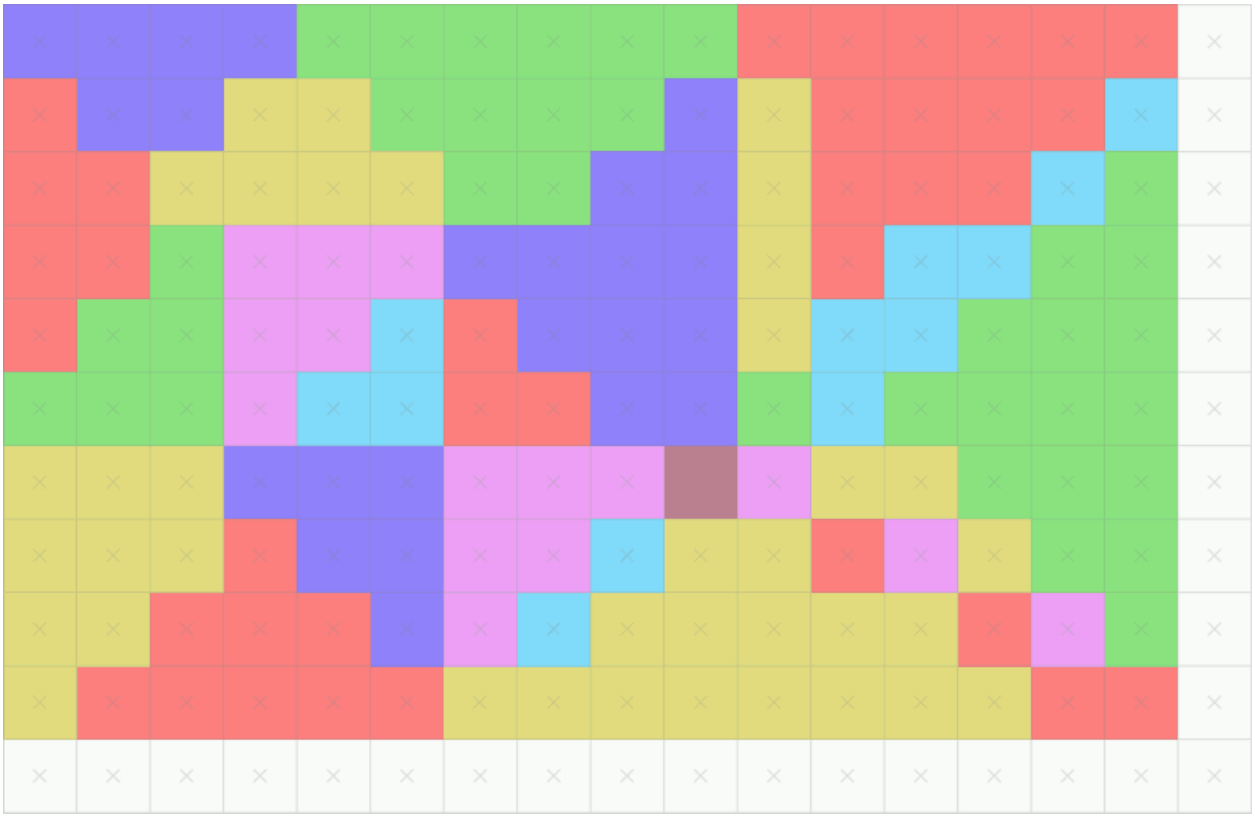
Introduce the **top-left rule**. A pixel's testpoint is considered to fall within a triangle's region if it lies on the *top edge* or the *left edge* of a triangle.

- A triangle's **top edge** is a horizontal edge which is above the other two edges.
- A triangle's **left edge** is a non-horizontal edge that is on the left side of the triangle.

Clearly, a triangle can either 0 or 1 top edges. A triangle can have 0 or 1 left edges.

When we draw triangles side by side, we often implicitly want them to be viewed as "connected", as a continuous surface. The top-left rule ensures that there is a uniform color applied to edges which are shared between two triangles. It also ensures no gaps between triangles.



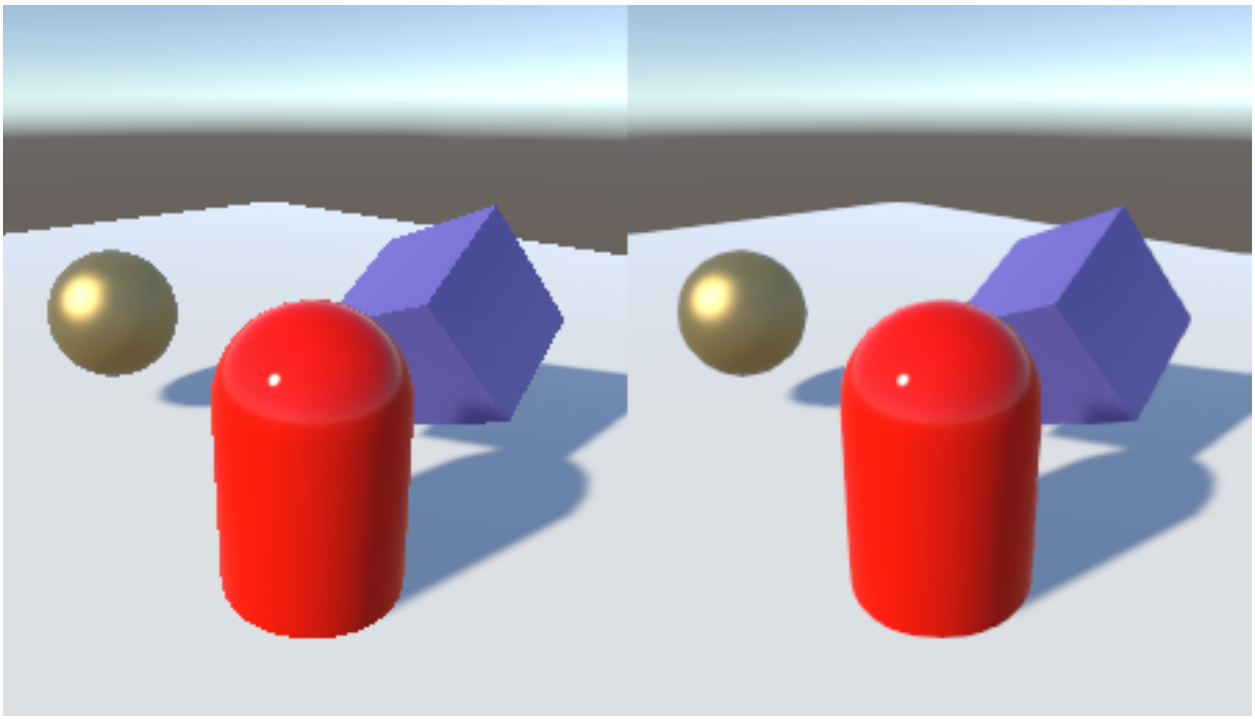


Let's make the top-left rule algorithmic.

1. If a test point is fully inside of a triangle, then it is covered by that triangle.
2. If a test point is on two edges of a triangle, and both edges are either top or left, then it is covered by that triangle.
3. If a test point is on exactly one edge of a triangle, and that edge is a top or left edge, then it is covered by that triangle.

6.2 Antialiasing

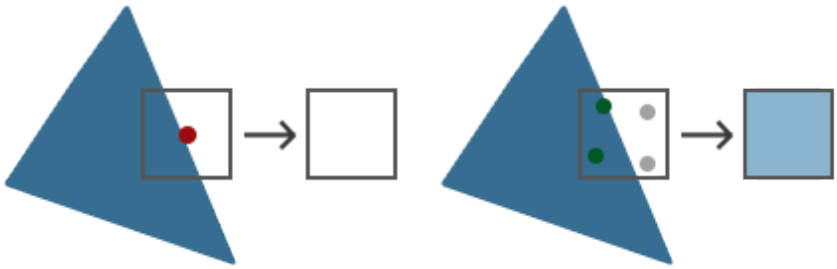
Alas, this discretization process can make things very "rugged", "jagged", "ugly". This ugliness is called aliasing. To get around it, we use **anti-aliasing**.

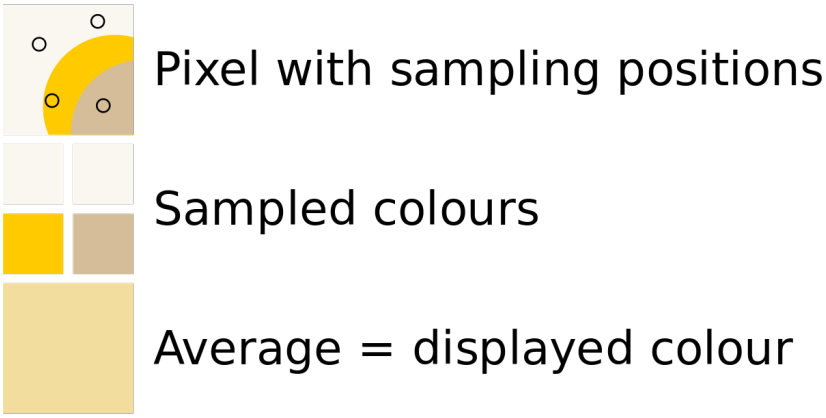


The most common form of anti-aliasing is **supersampling**. A special case of supersampling which is often used in practice is **multisampling**. The core concepts are the same. "In essence, multisampling is supersampling where the sample rate of the fragment shader is lower than the number of samples per pixel." (Fragment shader?, you ask. See Lecture 8.)

Consider that each pixel has N test points. For each test point covered by a primitive, add $\frac{1}{N}$ of that primitive's color to that pixel. Still use the top-left rule.

In other words, take the average color of all the colors sampled.

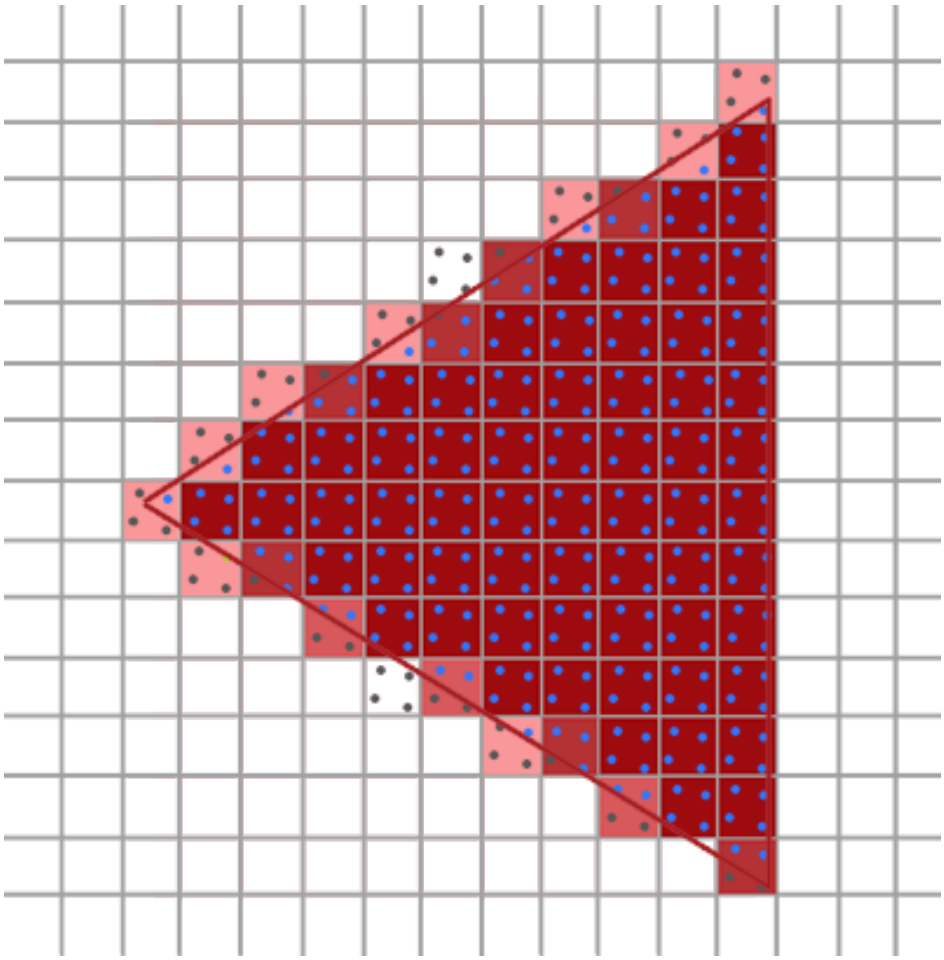




This is why you hear of "2x multisampling", "4x multisampling", etc. It's *N*x sampling. The more test points the more samples are made. But, anti-aliasing is expensive. You have to double, quadruple, octuple, etc., the number of samples needed to generate each pixel (in a naive implementation).

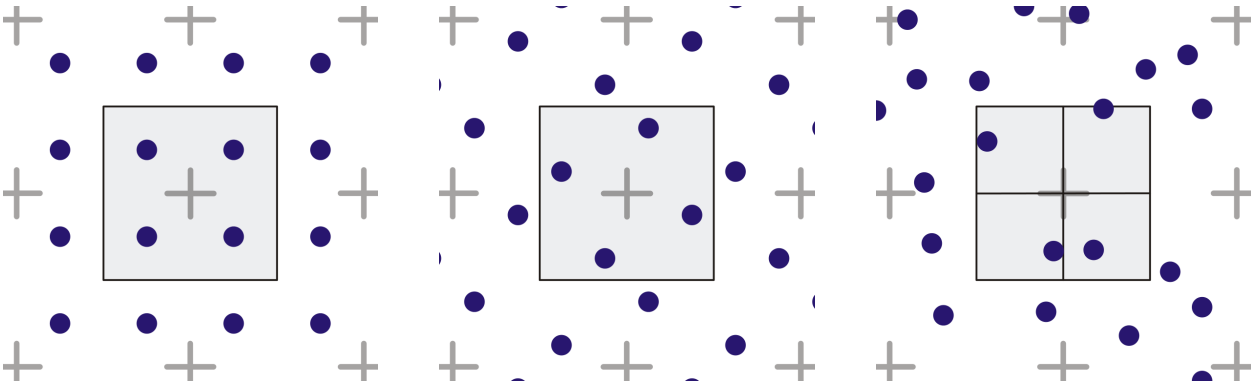
In OpenGL at least, it's easy to implement:

```
glfwWindowHint(GLFW_SAMPLES, 4);
glEnable(GL_MULTISAMPLE);
```



One parameter of supersampling is a the number of samples within each pixel, the number of test points. But what about their positions? Without supersampling the test point is usually the center of the pixel, as we have seen in the previous subsections.

One can obtain different **supersampling patterns** by changing the position of test points and each test point's "weight" in the overall averaged color computation. Common choices are *grid*, *rotated grid*, and *jitter* (structured random), shown respectively below. The rotated grid (**RGSS**) is the de facto standard for 4x antialiasing. Compare the three scenes below.



While grid is the easiest to implement and very fast in practice, it produces less than stellar results. Visually, our eyes are quite sensitive to “nearly horizontal” and “nearly vertical” lines. The regular sample points of the grid pattern are bad at handling such nearly horizontal/vertical lines.

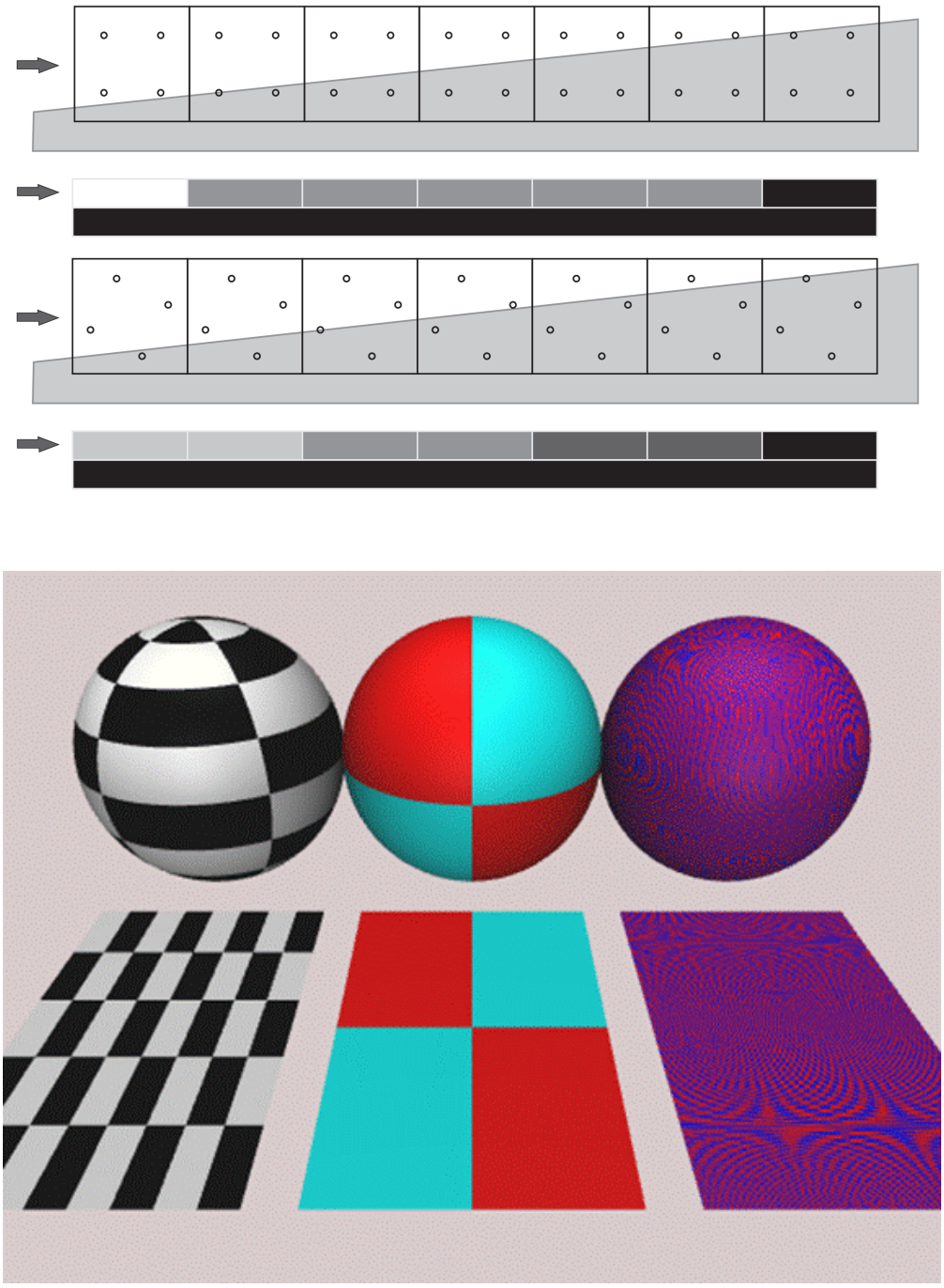


Figure 2: Grid pattern

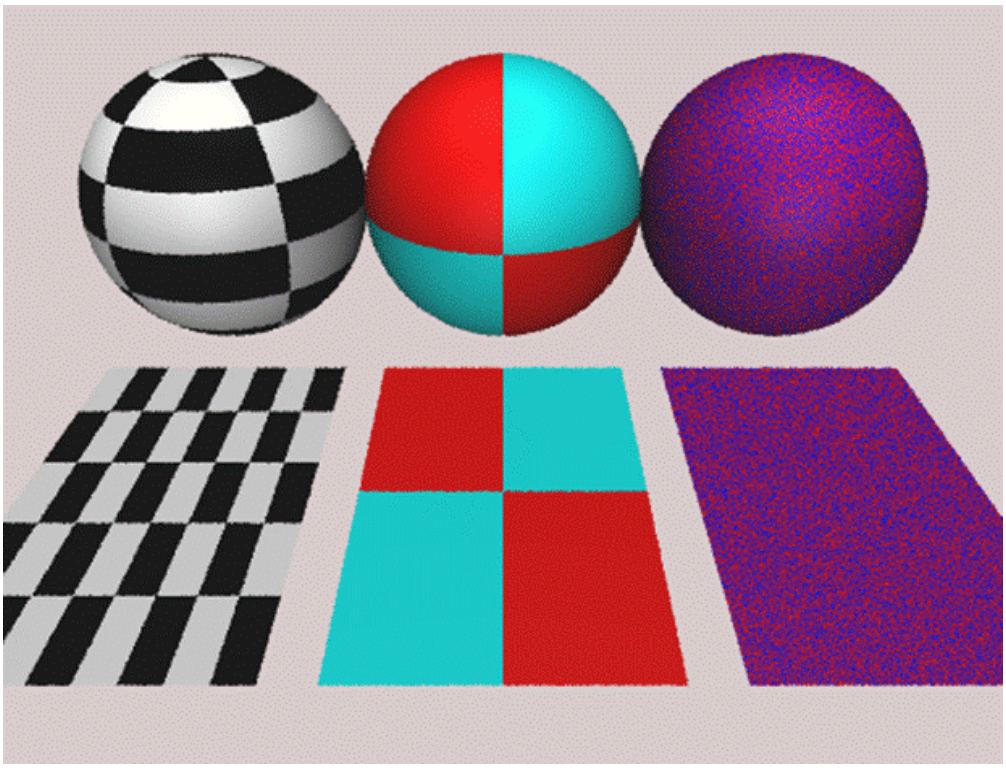


Figure 3: RGSS pattern

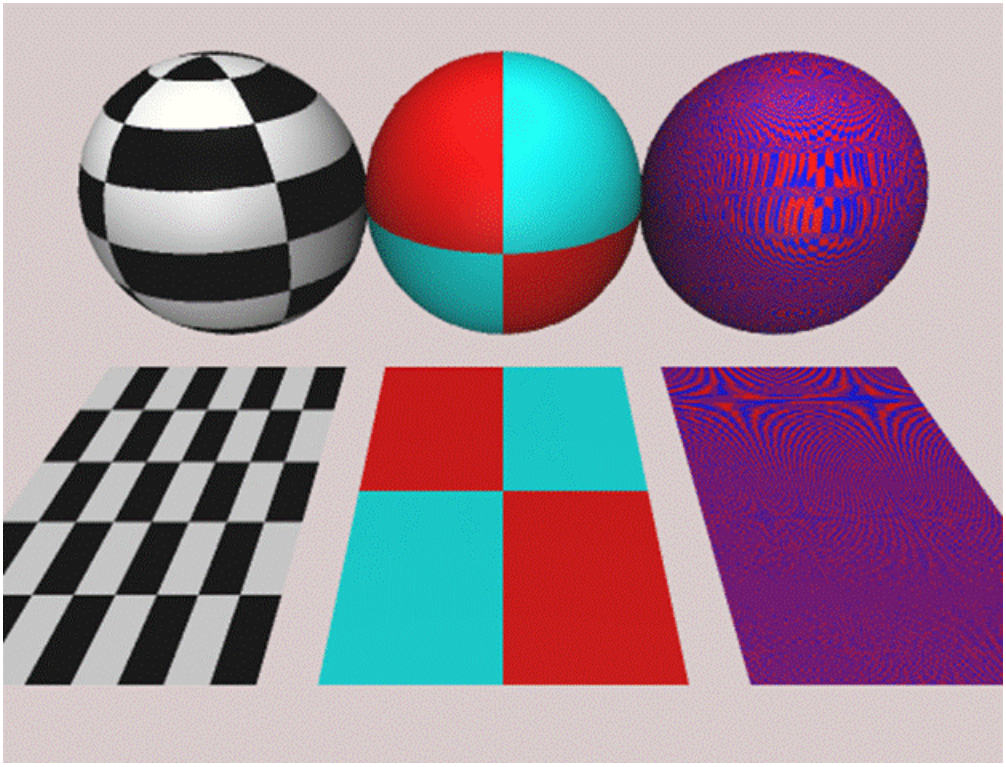


Figure 4: Jitter pattern