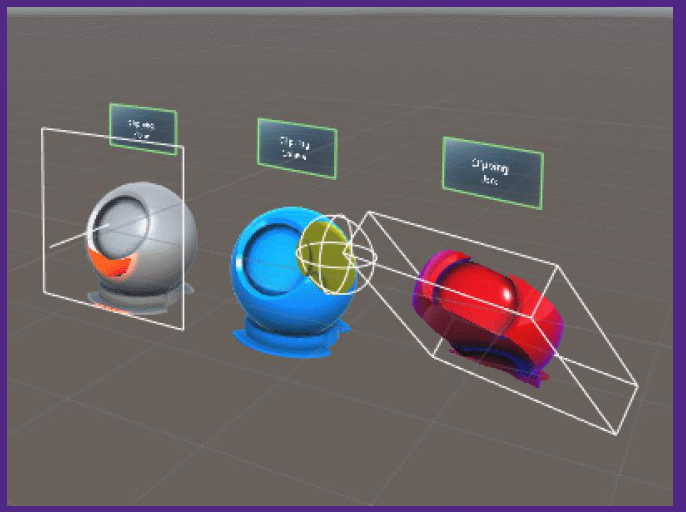
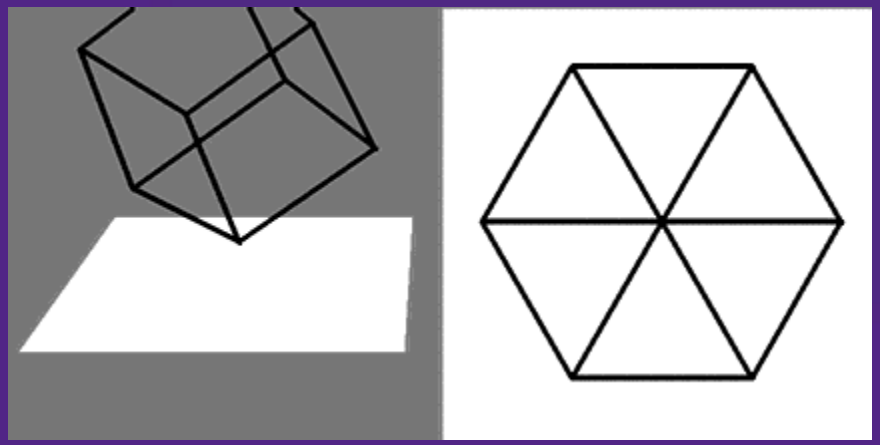
Computer Graphics I:
2D Clipping Algorithms



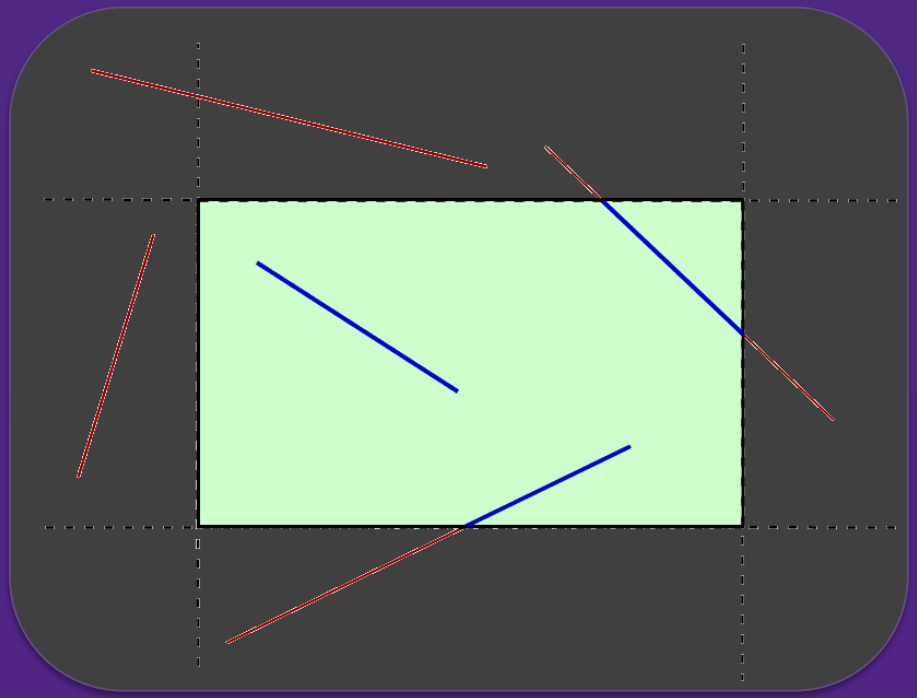


Review

- We specify the positions of vertices in world coordinates.
- It just so happens that, by default, world coordinates coincide with Normalized Device Coordinates because the projection matrix is the identity matrix.
- Recall we use glOrtho to specify the viewingvolume.
- It is the projection matrix used to transfer from the viewing volume to NDC). For example, if we specify a viewing volume where x ranges between 0 and 100, and y ranges between 0 and 100, then anything we draw at within those coordinates will be displayed on the screen.
- Anything outside will be clipped.

Line Clipping

Clipping is the process of excluding data from being drawn. It is a fundamental operation in graphics. Clipping is more than just excluding data. As the name suggests there is some "cutting" going on. If only part of primitive is outside the viewing volume (or viewport) we still want that part that is visible to be seen.



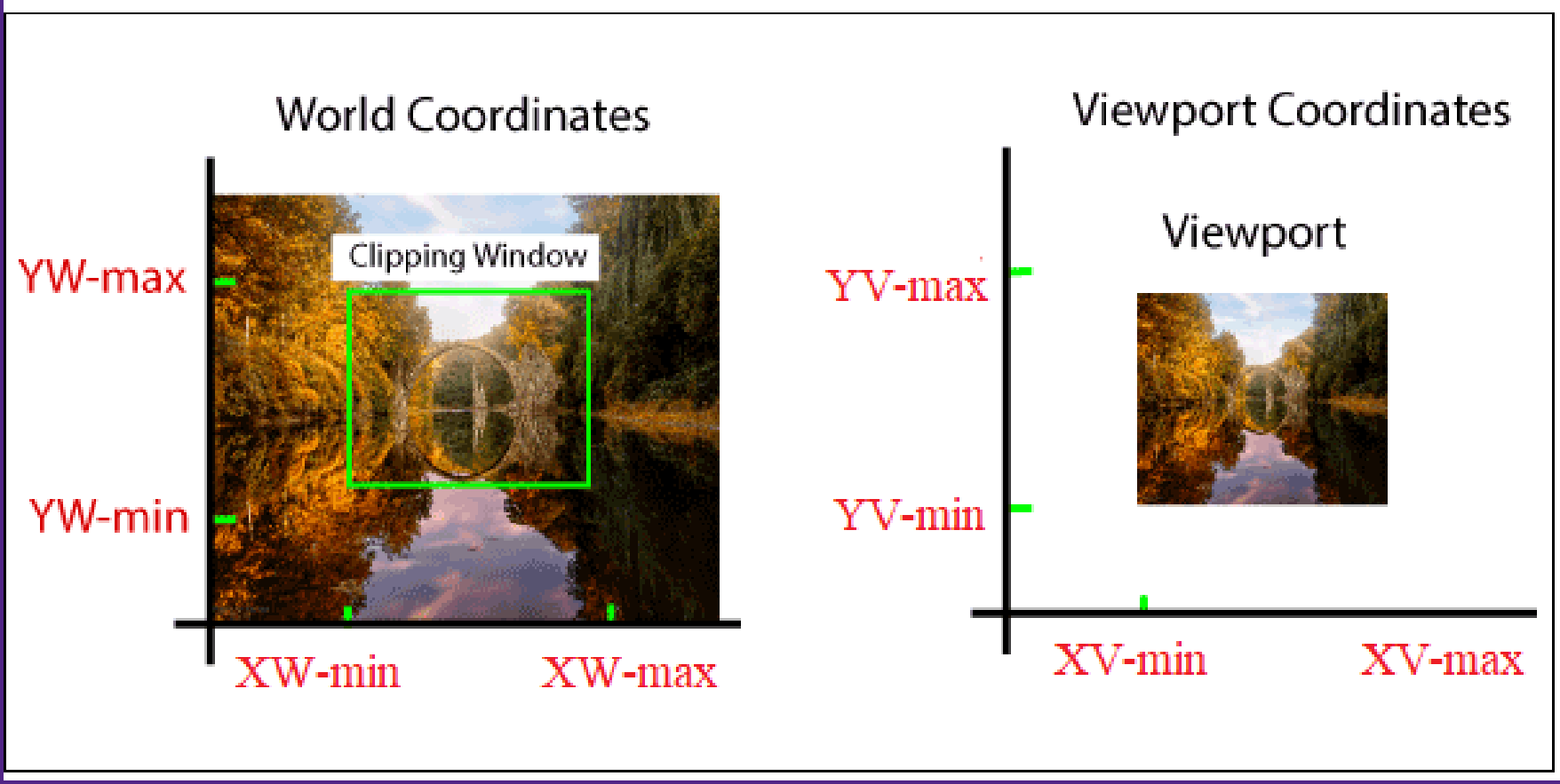
2D Clipping Algorithms

- The viewport is a region of the window in which graphics are displayed.
- The transformations from world coordinates, to camera coordinates, to window coordinates, and finally to viewport coordinates usually results in only a part of the scene being visible, and hence the need to clip lines to the viewport in 2D.
- Suppose a point (x, y) . It is inside the viewport if

$$Xmin \leq x \leq Xmax$$

$$Y\ min \leq y \leq Y\ max$$

2D Clipping Algorithms



Clipping Lines

- If both endpoints of a line lie inside the viewport, no clipping is necessary, as the line segment is completely visible.
- If only one endpoint is within the viewport, then we must clip the line at the intersection.
- If both endpoints are outside the viewport, then the line crosses two boundaries, or is completely outside the viewport.

Clipping Lines

- The key is to divide screen space into 9 areas, where only the center area is visible.
- The algorithm has three simple steps:
- 1. If the line is entirely inside the viewport, draw it completely.
- 2. If the line is entirely outside the viewport, draw nothing.
- 3. Otherwise, cut the line at one of the edges of viewport, keep the "inside" part of the cut line, and repeat the above with the new shorter line.

Clipping Lines

- Let's consider the first two cases.
- How do we determine if a line is completely outside or completely inside the viewing area?
- Line (segments) are described by two endpoints, say (x_0, y_0) and (x_1, y_1).
- The viewport can be described using its two diagonal corners: (x_{min}, y_{min}), (x_{max}, y_{max}).

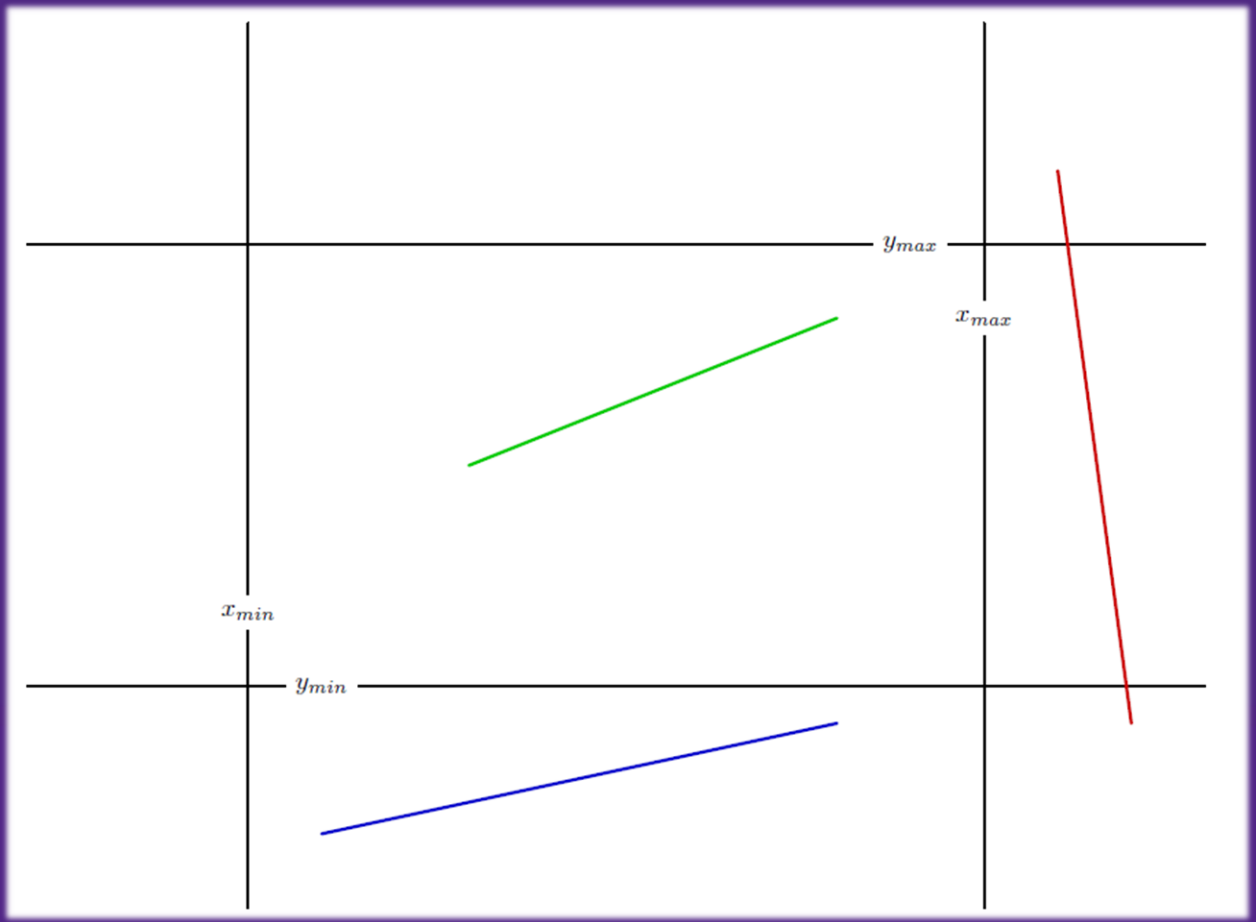
Clipping Lines

- For a point to be completely **inside** the viewport (e.g. green below):
- 1. $x_{min} \leq x_0 \leq x_{max}$, and
- 2. $y_{min} \leq y_0 \leq y_{max}$, and
- 3. $x_{min} \leq x_1 \leq x_{max}$, and
- 4. $y_{min} \leq y_2 \leq y_{max}$.

Clipping Lines

- For a point to necessarily be completely **outside** the viewport (e.g. red or blue below):
- 1. ($x_0 < x_{min}$ and $x_1 < x_{min}$) or
- 2. ($x_0 > x_{max}$ and $x_1 > x_{max}$) or
- 3. ($y_0 < y_{min}$ and $y_1 < y_{min}$) or
- 4. ($y_0 > y_{max}$ and $y_1 < y_{max}$) or

Clipping Lines



Clipping Lines

- Notice that the above test for testing "completely outside" may miss finding some lines which are completely outside.
- In particular, when one endpoint's x-coordinate is "inside" but its y-coordinate is "outside", and vice versa for the other end point. That's fine.
- It'll get chopped like any other line and then eventually found to be "trivially outside".
- The main idea to to chop once, then check using the above conditions if the chopped line is "trivially inside" or "trivially outside".
- Otherwise, chop it again and repeat the trivial checks.
  - Cross course connection, what kind of algorithm paradigm is this like?

Clipping Lines

- To do the chopping and the cutting, we have several choices. A classic choice is TBRL. which uses the "top" y_{max} line as the knife, then uses the bottom y_{min} line as the knife, then x_{max}, then x_{min}.
- We can use classic $y = mx + b$ to determine the new endpoints after cutting.
- The key is that, when being cut by "top", the endpoint's new y coordinate is y_{max}.
- When cut by "bottom" the endpoint's new y is y_{min}.
- Same goes for left and right, but for x.

Clipping Lines

Then, we can use the line's slope to determine the corresponding x coordinate (when cut by top or bottom) or y coordinate (when cut by right or left).

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

Then, from (x_0, y_0), following along a straight line of slope m, and ending at y_{max} must have corresponding x coordinate x_{new}:

$$\frac{y_1 - y_0}{x_1 - x_0} = m = \frac{y_{max} - y_0}{x_{new} - x_0}$$

$$(x_{new} - x_0)(y_1 - y_0) = (y_{max} - y_0)(x_1 - x_0)$$

$$x_{new} - x_0 = (y_{max} - y_0)(x_1 - x_0)/(y_1 - y_0)$$

$$x_{new} = x_0 + (y_{max} - y_0)(x_1 - x_0)/(y_1 - y_0)$$

The the almost same formula can be used to find x_{new} at y_{min} by just replacing y_{max} with y_{min}.

Clipping Lines

When we are cutting along right or left, the new x-coordinate will be x_{max} or x_{min}, respectively. We can then find y_{new} using the same slope formula:

$$\frac{y_1 - y_0}{x_1 - x_0} = m = \frac{y_{new} - y_0}{x_{max} - x_0}$$

$$(x_{max} - x_0)(y_1 - y_0) = (y_{new} - y_0)(x_1 - x_0)$$

$$y_{new} - y_0 = (x_{max} - x_0)(y_1 - y_0)/(x_1 - x_0)$$

$$y_{new} = y_0 + (x_{max} - x_0)(y_1 - y_0)/(x_1 - x_0)$$

Cohen-Sutherland's Line Clipping Algorithm

- The viewing space is divided into nine encoded regions:

1001	**1000**	**1010**
0001	**0000**	**0010**
0101	**0100**	**0110**

- For each endpoint of a line segment, we assign a 4-bit code following the rules below (the code is formed in the following way: (bit 1, bit 2, bit 3, bit 4):
  - **Bit 1 is 1 if $x < X_{min}$**
  - **Bit 2 is 1 if $x > X_{max}$**
  - **Bit 3 is 1 if $y < Y_{min}$**
  - **Bit 4 is 1 if $y > Y_{max}$**
- Hence, if (x_1, y_1) and (x_2, y_2) have code 0000 then both end points are inside. If both endpoints have a 1 in the same bit position, then the line segment is entirely outside. Performing a logical AND on both codes and obtaining something different from 0000 indicates that the line is outside. Otherwise, the line segment must be checked for intersections.

Calculating Intersections

- With a vertical boundary, x is either X_{min} or X_{max} . Hence:

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(X_{min} - x_1)$$

or

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(X_{max} - x_1)$$

Calculating Intersections

- With a horizontal boundary, y is either Y_{min} or Y_{max} . Hence:

$$x = x_1 + \left(\frac{x_2 - x_1}{y_2 - y_1}\right)(Y_{min} - y_1)$$

or

$$x = x_1 + \left(\frac{x_2 - x_1}{y_2 - y_1}\right)(Y_{max} - y_1)$$

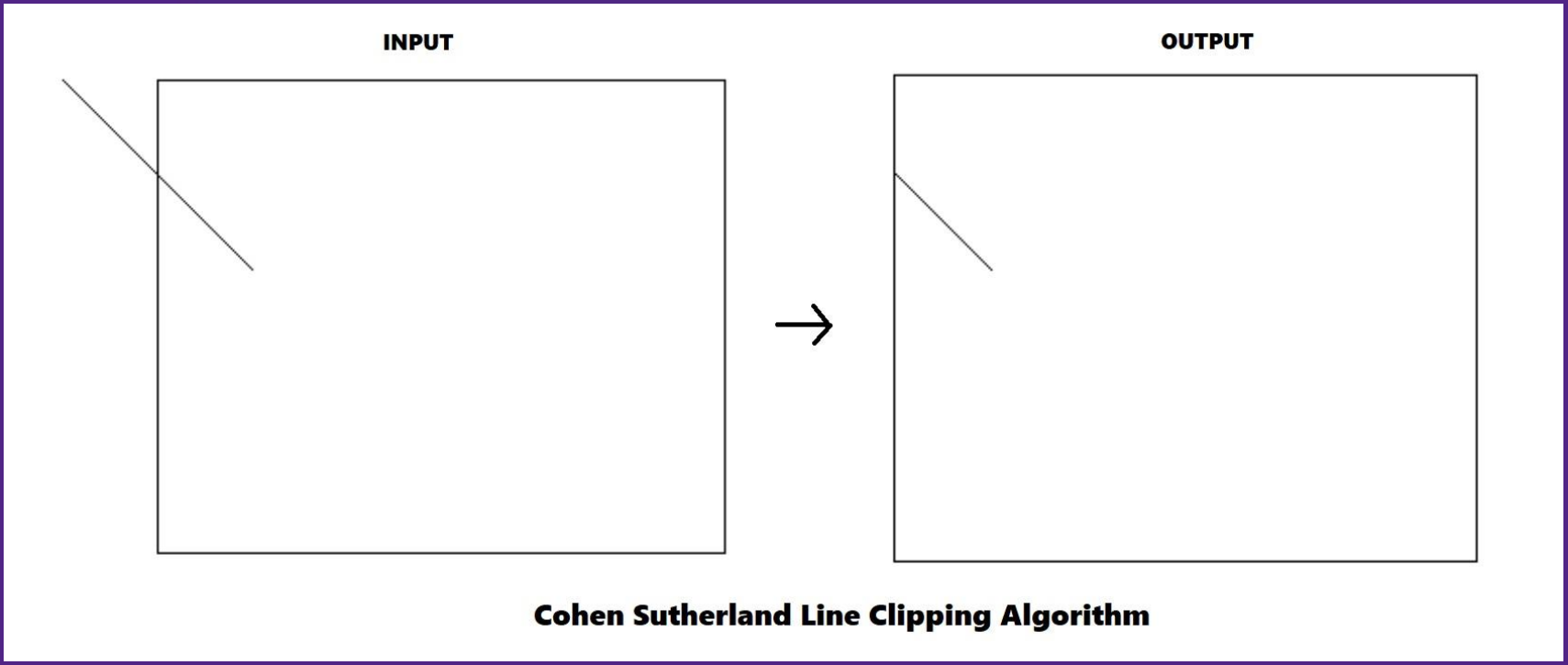
Calculating Intersections - Algorithm

- The inputs to the algorithm are the two endpoints of a line segment, and the attributes of the viewport.
- The output is the clipped line segment:

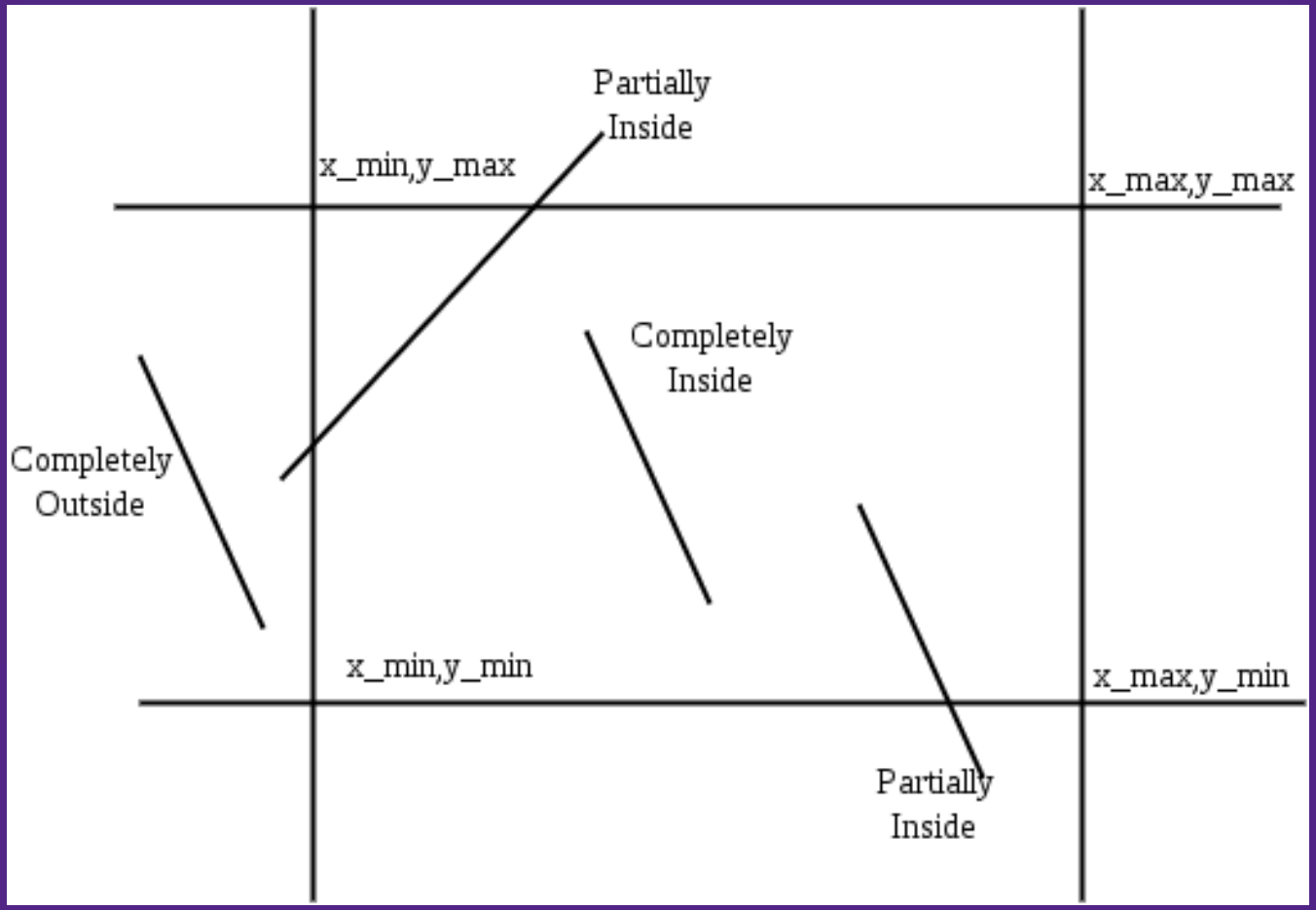
```
if (p1 and p2 inside)
    output p1 and p2 and return
else
    while (code of p1 != 0000) do
        if (p1 is to the left)
            p1 = intersection of p1p2 and x=Xmin
        else if ( p1 is to the right)
            p1 = intersection of p1p2 and x=Xmax
        else if ( p1 is below)
            p1 = intersection of p1p2 and y = Ymin
        else if ( p1 is above)
            p1 = intersection of p1p2 and y = Ymax
        update code for p1
    end while
repeat for p2
output p1 and p2 and return
```

```
//let xmin xmax, ymin, ymax be defined as globals
bool CS_clip (float& x0, float& y0, float& x1, float &y1) {
    while(1) {
        if (line totally inside viewport) {
            return true;
        } else if (line totally outside) {
            return false;
        } else if ((x0,y0) is outside) {

            if (y0 > ymax)  { // top
                x0 = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y0 = ymax;
            } else if (y0 < ymin) { // bottom
                x0 = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y0 = ymin;
            } else if (x0 > xmax) { //right
                y0 = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x0 = xmax;
            } else { //left
                x0 = xmin;
                y0 = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
            }

        } else ((x1,y1) is outside) {
            //repeat above but for (x1,y1)
        }
    }
}
```

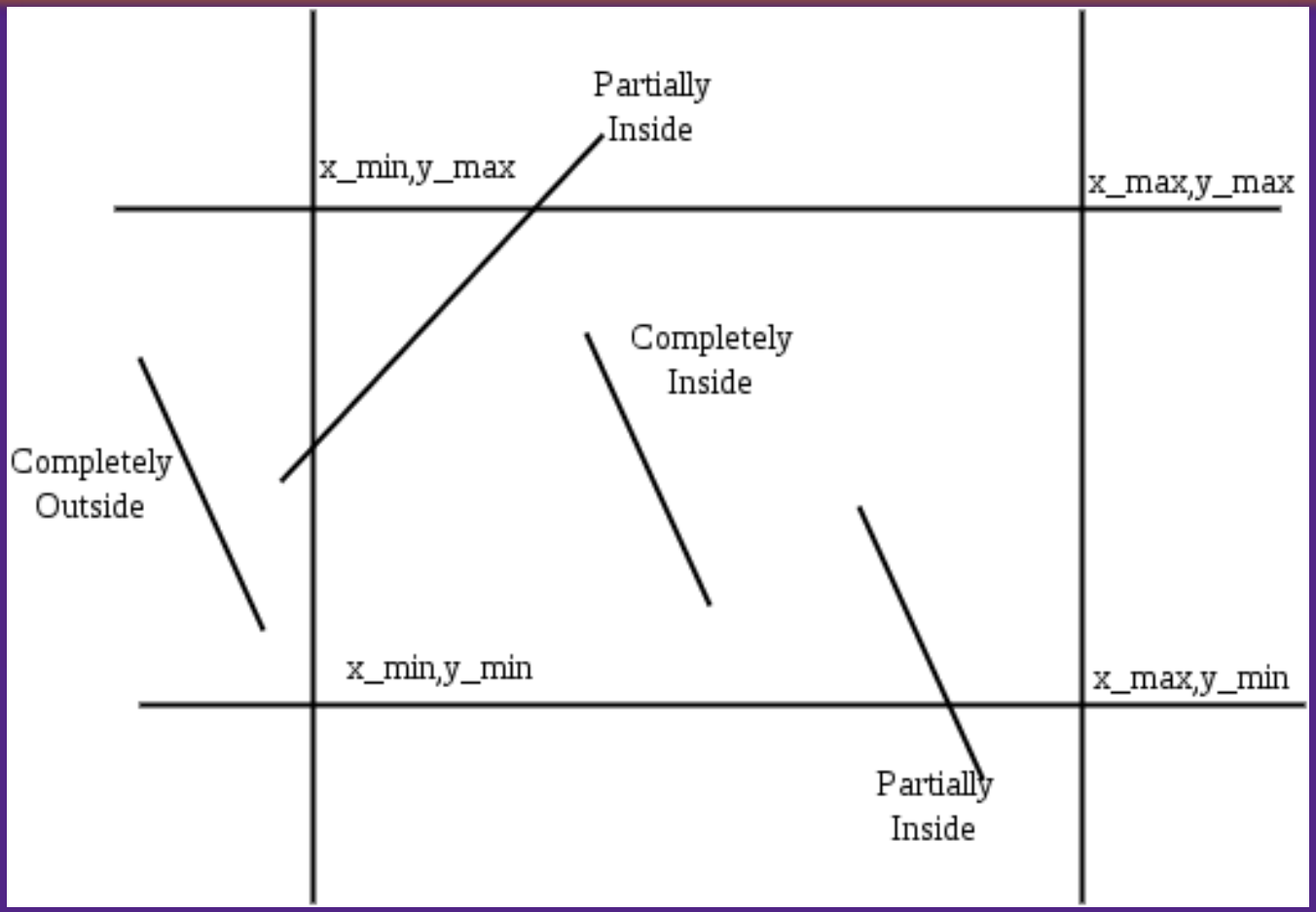
Calculating Intersections - Algorithm



Calculating Intersections - Algorithm



Thinking Question – Limitations of this algorithm?



Midpoint Division Method

- This technique consists of locating intersections without computing them with a binary search.
- First, compute the midpoint of the segment, and divide the segment into two as per the midpoint.
- For these two segments, test if they are totally inside or totally outside of the viewport and either accept or reject the segments accordingly.
- Repeat for each segment that cannot be either accepted or rejected.

Liang and Barsky's Clipping Algorithm

- This algorithm uses the parametric form of line equations, written as:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} u = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} u$$

- where u∈[0,1]. Using this particular form of equations then we can write that when the following inequalities are satisfied

$$X_{min} \leq x_1 + \Delta x u \leq X_{max}$$

$$Y_{min} \leq y_1 + \Delta y u \leq Y_{max}$$

- then the line segment lies completely within the window.
- These inequalities can be written as $p_k u \leq q_k$

Liang and Barsky's Clipping Algorithm

- for k=1,2,3,4:

k=1	$p_1=-\Delta x$	$q_1=x_1-X_{min}$	Left boundary
k=2	$p_2=\Delta x$	$q_2=X_{max}-x_1$	Right boundary
k=3	$p_3=-\Delta y$	$q_3=y_1-Y_{min}$	Bottom boundary
k=4	$p_4=\Delta y$	$q_4=Y_{max}-y_1$	Top boundary

- If p_k=0 then the line segment is parallel to the k^{th} boundary:
- $p_1=-\Delta x=0$ then the line is vertical and parallel to the left and right boundaries
- in addition to this , if $q_1<0$ or $q_2<0$, then the line segment is completely outside the window

Liang and Barsky's Clipping Algorithm

- Hence, if (p_1=0)∧(q_1<0∨q_2<0) the line segment is completely outside.
- The same rules apply for p_3 and we can write that if (p_3=0)∧(q_3<0∨q_4<0) then the segment is also outside.
- In general, if q_k≥0 then the line is inside the k^{th} boundary.
- In cases when the line segment is not parallel to any of the view port's borders, then if p_k<0 the infinite extension of the line segment proceeds from outside to inside the infinite extension of the k^{th} window boundary.
- The opposite situation occurs when p_k>0 .

Liang and Barsky's Clipping Algorithm

- Consequently, when $p_k \neq 0$, we compute the value of u that yields the intersection of the extended line segment with the extended k^{th} boundary. The value of u is given by equating the initial inequalities:

$$p_k u = q_k$$

- and thus, the intersection of the extended line segment with the kth boundary is simply given by:

$$u = \frac{q_k}{p_k}$$

The Algorithm for Liang and Barsky

Computing u_1:

for all $p_k < 0$

$r_k = q_k / p_k$

$u_1 = \max\{0, r_k\}$

Computing u_2:

for all $p_k > 0$

$r_k = q_k / p_k$

$u_2 = \min\{1, r_k\}$

The Algorithm for Liang and Barsky

compute p_k and q_k, for k=1,2,3,4

if [(p_1=0)∧(q_1< 0 ∨ q_2<0)] ∨ [(p_3=0)∧(q_3<0∨q_4<0)]

reject the segment

else

compute u_1

compute u_2

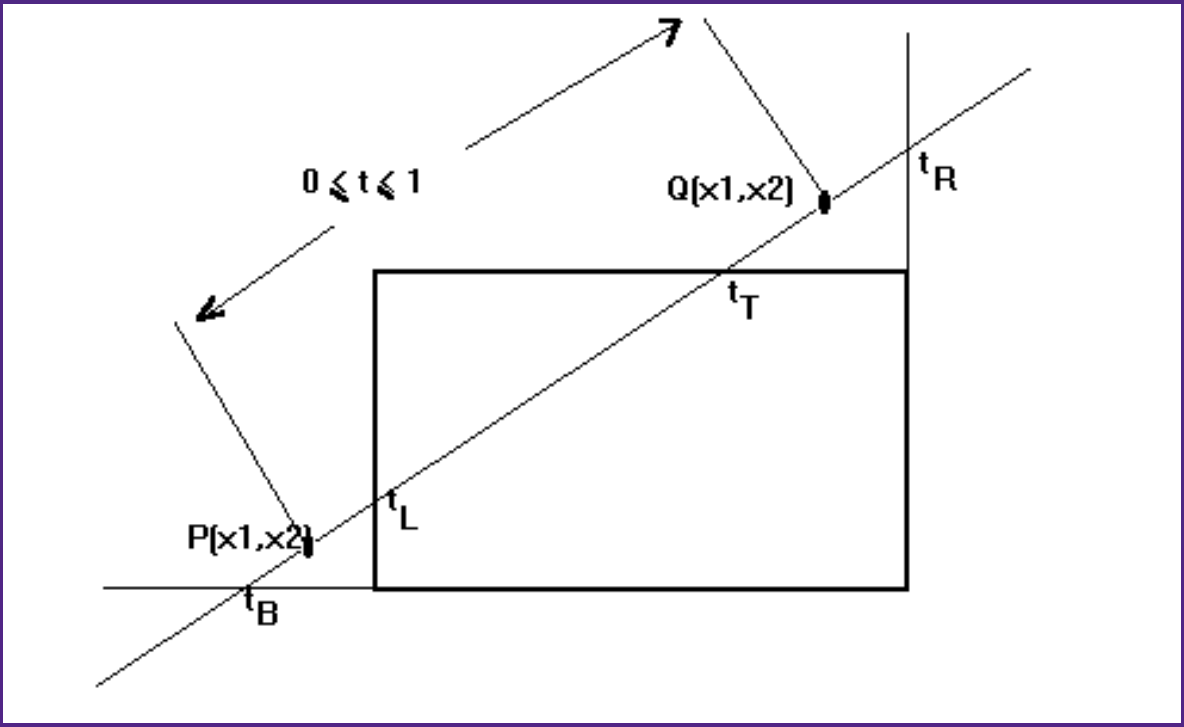
if u_1>u_2

reject the segment

else

clipped line segment is (x_1+Δ xu_1, y_1+Δ yu_1) , (x_1+Δ xu_2, y_1+Δ yu_2)

Liang and Barsky's Clipping Algorithm



Polygon Clipping

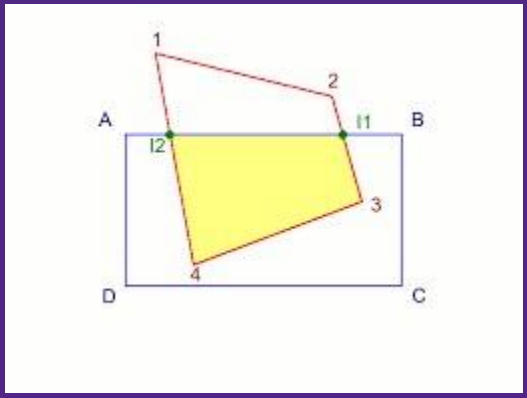
- Polygons need to be clipped when objects to render are made with them.
- Polygons are represented as ordered sets of vertices.
- An intuitive approach to clip polygons is to trace the vertices that are completely inside the viewport, and get rid of those that are completely outside.
- For all other vertices, and using their parametric representation, we can determine if they are tracked from the outside toward the inside of the area of the viewport, and conversely.

Polygon Clipping

- So we can clip lines. Cool.
- What about two-dimensional primitives?
- What about triangles? Sutherland strikes again.
- But, honestly, graphics libraries are sophisticated enough now that you never need to know how this works.
- If you ever need to implement this algorithm yourself, you're certainly doing low-level graphics programming and can figure out this algorithm for yourself. For reference, see this wiki article.

Polygon Clipping

- Below is a simple example where the intersections of the vertices are all within the same boundary.
- All that is needed after clipping is to create a vertex joining the two intersections, and the polygon is clipped.
- However, this simple method will not work when the intersections are not all with the same boundary.
- When this is the case, the polygon must be adjusted to include the part of the viewport that finds itself inside it.



Intersection Between Segments

- In order to perform this type of polygon clipping, we must first know how to compute intersections when the segments are given in parametric form. Consider:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} u$$

- and let's find the intersection between $\vec{p}_1=(x_1, y_1),(x_2, y_2)$ and $\vec{p}_2=(x_3, y_3),(x_4, y_4)$

$$\vec{p}_1 = \vec{x}_1+\Delta \vec{x}_1 u$$
$$\vec{p}_2 = \vec{x}_2+\Delta \vec{x}_2 v$$

- and find (u,v) values such that $\vec{p}_1 = \vec{p}_2$.

Intersection Between Segments

- We can write the equality from last slide as:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} u = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} + \begin{pmatrix} x_4 - x_3 \\ y_4 - y_3 \end{pmatrix} v$$

- And rearrange it into the following:

$$\begin{pmatrix} x_2 - x_1 & x_4 - x_3 \\ y_2 - y_1 & y_4 - y_3 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} x_3 - x_1 \\ y_3 - y_1 \end{pmatrix}$$

- This is then simply a 2 by 2 linear system of equations that need to be solved.
- If the lines are not parallel, it has a solution for $(u,v)^T$ which yields the intersection point.

Raster Scan Polygon Filling

- Rendering polygons implies filling them with color, texture, etc.
- Here is a simple scan-line algorithm to do just this, with the following steps.
- The algorithm accepts a list of points describing the polygon and a color to fill it with:

Determine the number of scan lines. The first scan line starts at the minimum y value of all points while the last scan line is at the maximum y value of all points.

Create a segment list containing all the segments from the polygon, trace the horizontal edges and remove them from the list.

<Continued>

Raster Scan Polygon Filling

For each scan line y do:

Create a list of active segments from the segment list. These are the segments that make an intersection with the scan line y .

For each segment in the active list do:

(1) If the segment is not vertical, compute its intersection with the scan line y as I = (y-b)/m, where m is the slope of the segment, b is the y coordinate of the active segment's first end-point to which we subtract m times the x coordinate of the same active segment's first end-point

(2) Else, compute the intersection I as the x coordinate of the active segment's first end-point.

(3) If the scan line y intersects with the active segment's first endpoint, then

(3.1)Set y_o to the y coordinate of the point immediately preceding the active segment's first point within the active list.

(3.2) Set y_1 to the y coordinate of the point that immediately follows the active segment's first point (this is the active segment's second point)

(3.3) If $y_0 < y < y_1$ or $y_0 > y > y_1$, add I to the intersection list.

(3.4) Else, add I twice to the intersection list

Raster Scan Polygon Filling

For each scan line y do:

Create a list of active segments from the segment list. These are the segments that make an intersection with the scan line y .

For each segment in the active list do:

(4) Else if the y coordinate of the segment's second point does not intersect with the scan line y, then add I to the intersection list.

Sort the intersection list in ascending order

Trace a horizontal line segment on the scan line y for each contiguous intersection pair from the intersection list.

(Done)

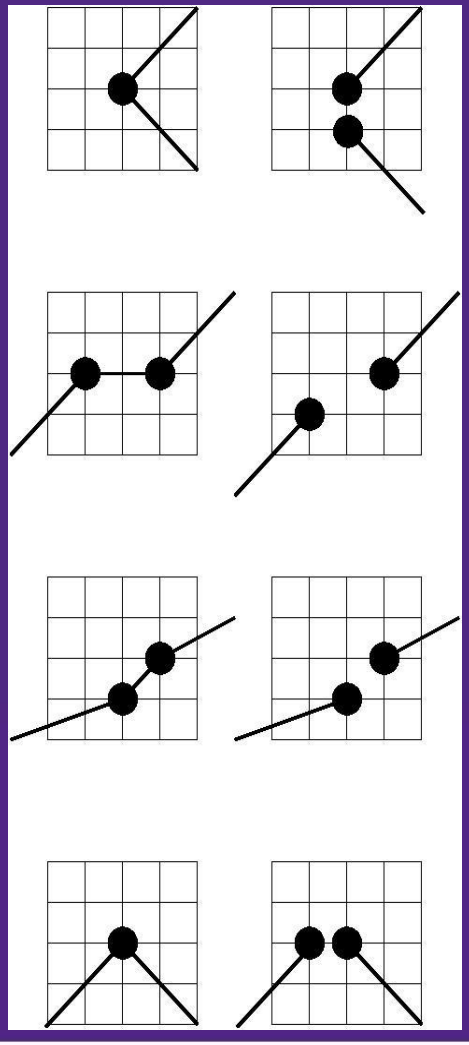
To implement this efficiently, it is important to observe that each scan line has a y value one pixel lower than the previous values, meaning:

$y_{i+1}=y_i - 1$, slope of segment is $m = \frac{y_{i+1}-y_i}{x_{i+1}-x_i}$, which implies $x_{i+1}= x_i - (1/m)$

Adjusting Segment Endpoints

- Instead of dealing with the number of intersections where the segments meet on the scan lines, we could also simply perform minor adjustments to the points of the polygons to ensure that each time an intersection is encountered, we either start or stop turning on pixels.
- On the following page are displayed all the cases (to the left) and how they should be dealt with (to the right).
- While this technique will slightly modify the appearance of the polygon at low resolutions, it is virtually undetectable with current resolutions and speeds up the scan-line algorithm.

Adjusting Segment Endpoints



Back to Transforms

- We know about the math of affine transforms. Now let's see them in practice.
- Recall that one can view affine transforms as transforming the coordinate system, and then vertices are drawn with respect to this modified coordinate system. (The opposite view is that the transforms manipulate the positions of the vertices themselves).
- Consider a simple triangle with vertices (0, 1), (1, 0), (−1, 0). With no other modifications, this will be drawn in NDC and thus span the width of the entire window and the top-half.

Back to Transforms

- We saw previously that we can use glOrtho to define the viewing volume in world coordinates which then get mapped to NDC.
- With a viewing volume spanning from −10 to 10 in x and y, the same triangle is not just a small figure with width 2 in a viewing volume of width 20.
- But how do we move the triangle around the viewing volume?
- We could manually compute transformation matrices to get the desired result and then multiply these matrices against each vertex's position. But, this is rather silly.

Back to Transforms

- Say we want to draw the triangle so that it's upper point is at (5, 6) in the world.
- Since this point is originally at (0, 1), the corresponding displacement vector is (5, 5).
- We translate the coordinate system by (5, 5), draw the triangle with its pre-defined vertex positions, and then **undo** the translation to get the original coordinate system back.

Back to Transforms

In code we can do this using `glTranslatef(x,y,z)`

```
1  glMatrixMode(GL_PROJECTION);
2  glLoadIdentity();
3  glOrtho(-10, 10, -10, 10, -1, 1);
4
5  glTranslatef(5, 5, 0);
6  glBegin(GL_TRIANGLES);
7      glVertex2f(0.0f, 1.0f);
8      glVertex2f(-1.0f, 0.0f);
9      glVertex2f(1.0f, 0.0f);
10 glEnd();
11 glTranslatef(-5, -5, 0);
```

Back to Transforms

- Now, remember two things:
- 1. OpenGL is a global state machine
- 2. glOrtho, and all transforms, multiply the current matrix by their corresponding transformation matrix
- So, in the above code we are modifying the projection matrix to translate the triangle... not a great

  (although it will get you the correct result.. for now).
- Instead, let's use a new matrix: GL_MODELVIEW.

Back to Transforms

```
1  glMatrixMode(GL_PROJECTION);
2  glLoadIdentity();
3  glOrtho(-10, 10, -10, 10, -1, 1);
4
5  glMatrixMode(GL_MODELVIEW);
6  glLoadIdentity();
7  glTranslatef(5, 5, 0);
8  glBegin(GL_TRIANGLES);
9      glVertex2f(0.0f, 1.0f);
10     glVertex2f(-1.0f, 0.0f);
11     glVertex2f(1.0f, 0.0f);
12 glEnd();
13 glTranslatef(-5, -5, 0);
```

Back to Transforms

- In OpenGL the modelview matrix is applied before the projection matrix.

$$v_{ndc} = PVv_{local}$$

- The model view matrix transforms an object's local coordinates to world coordinates.
- Or, another way of thinking about it, it (temporarily) moves around world coordinate system.
- To see the power of the model view matrix, let's try drawing two triangles.
- One whose tip is at (6, 5) and another whose tip is at (−4, −5).

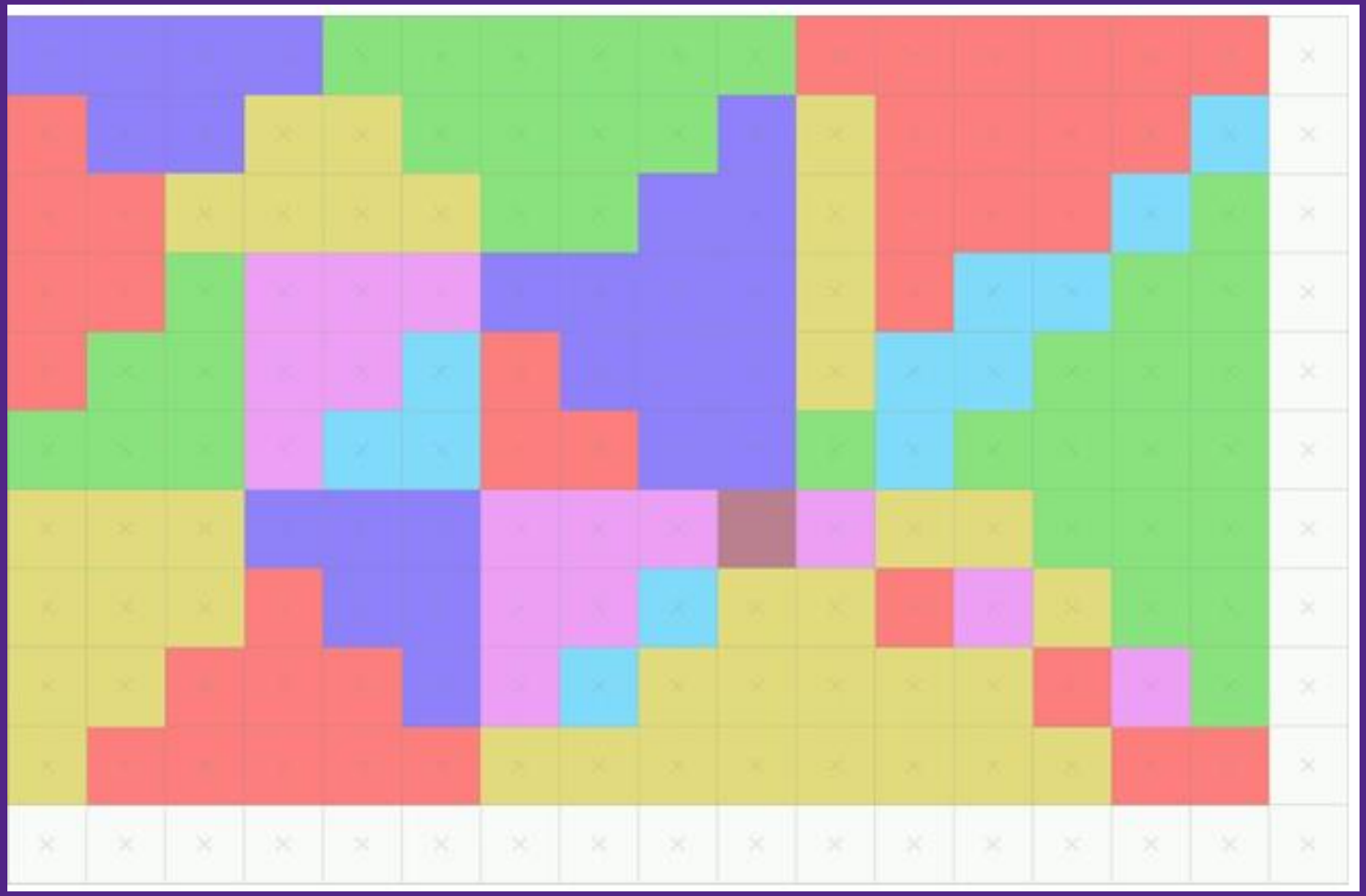
Back to Transforms

```
1  void drawTriangle() {
2      glBegin(GL_TRIANGLES);
3          glVertex2f(0.0f, 1.0f);
4          glVertex2f(-1.0f, 0.0f);
5          glVertex2f(1.0f, 0.0f);
6      glEnd();
7  }
8
9  glMatrixMode(GL_PROJECTION);
10 glLoadIdentity();
11 glOrtho(-10, 10, -10, 10, -1, 1);
12
13 glMatrixMode(GL_MODELVIEW);
14 glLoadIdentity();
15 glTranslatef(5, 5, 0);
16 drawTriangle();
17
18 glLoadIdentity();
19 glTranslatef(-5, -5, 0);
20 drawTriangle();
```

Back to Transforms

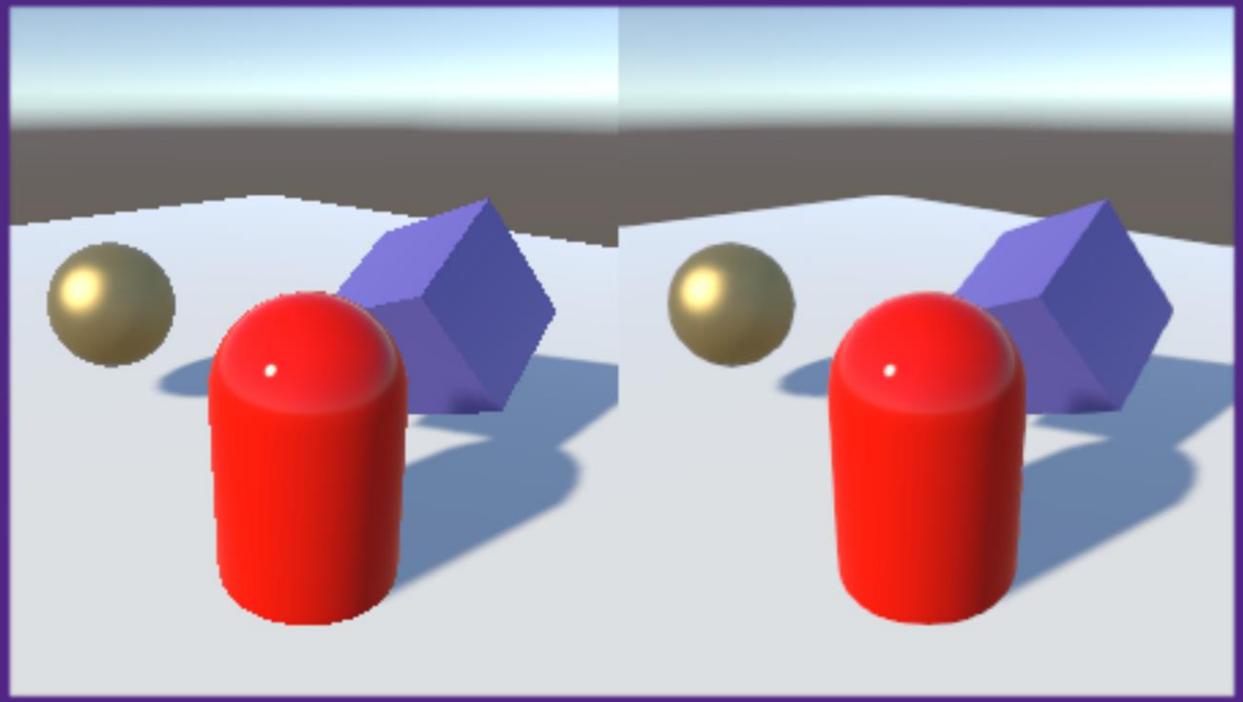
- So, we can always move around the coordinate system and draw vertices with respect to the moved coordinate system.
- We don't have to modify any of the object's vertices directly.
- We can apply many transformations in this way:
- •glTranslatef
- •glRotatef
- •glScalef

Triangle Rasterization



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.



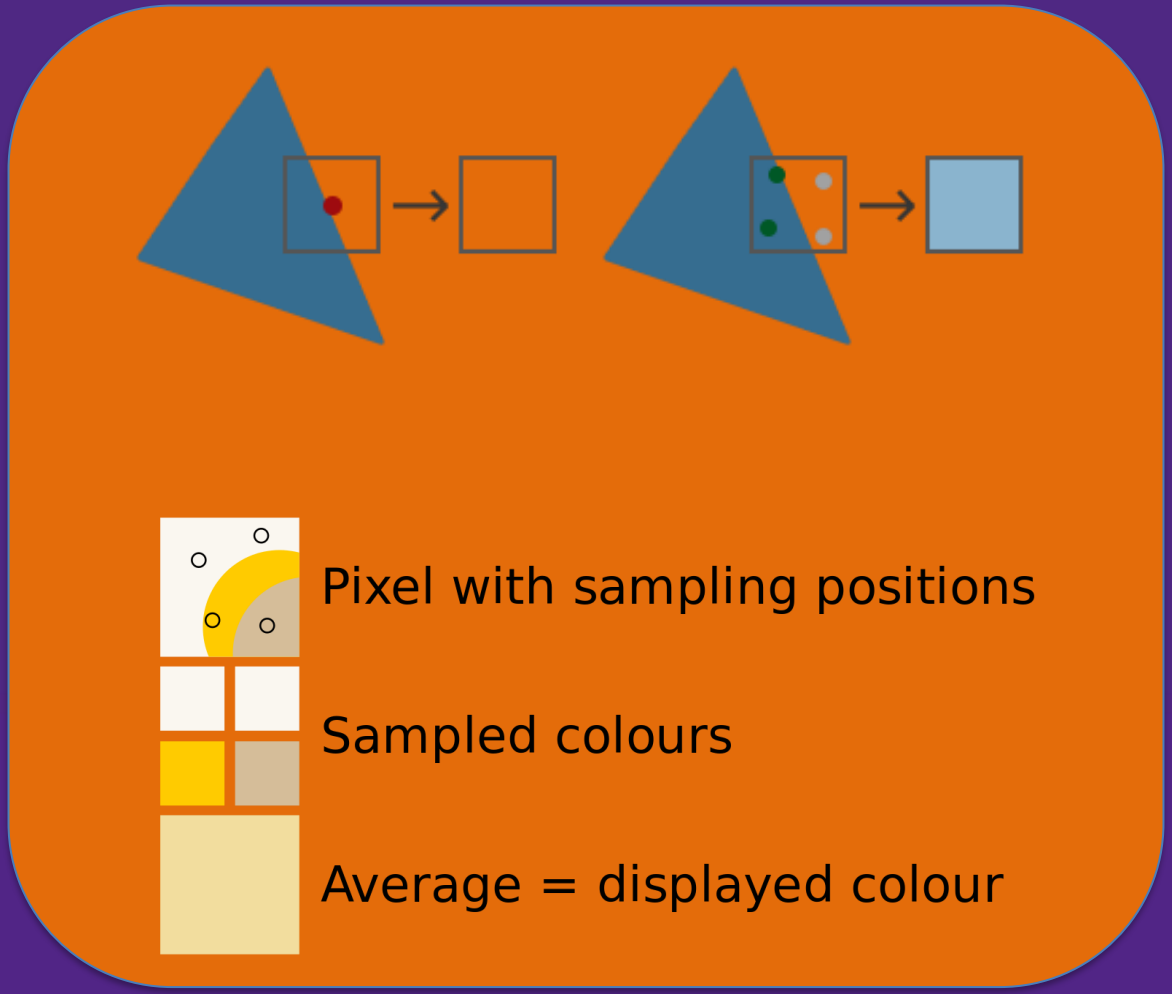
Antialiasing

- 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? Coming soon in Lecture 8.)

- Consider that each pixel has N test points.
- For each test point covered by a primitive, add N1 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.

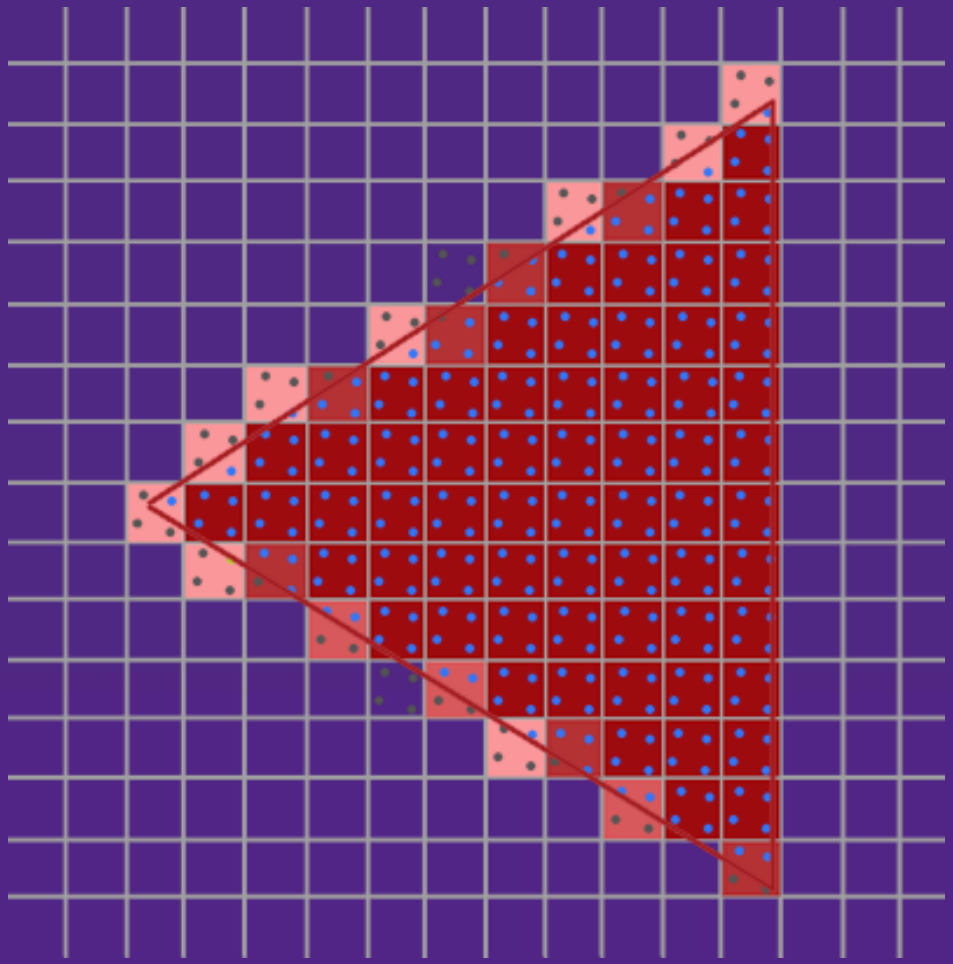
Antialiasing



Antialiasing

- This is why you hear of “2x multisampling”, “4x multisampling”, etc. It’s Nx 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:
- glfWindowHint(GLFW_SAMPLES,4);
- glEnable(GL_MULTISAMPLE);

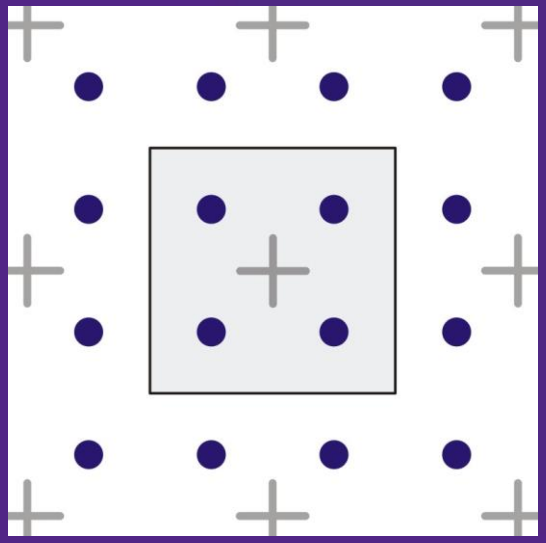
Antialiasing

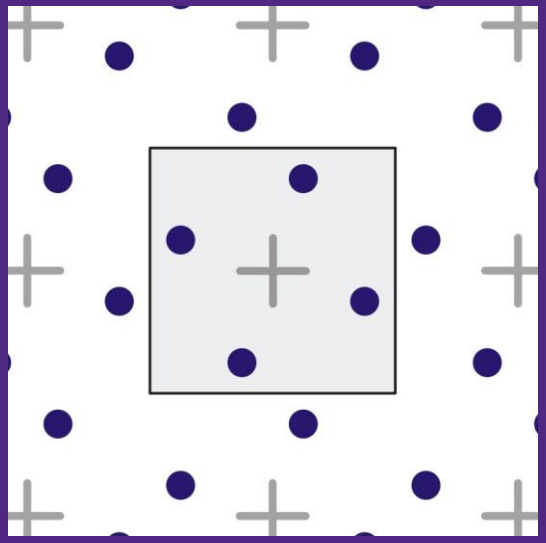


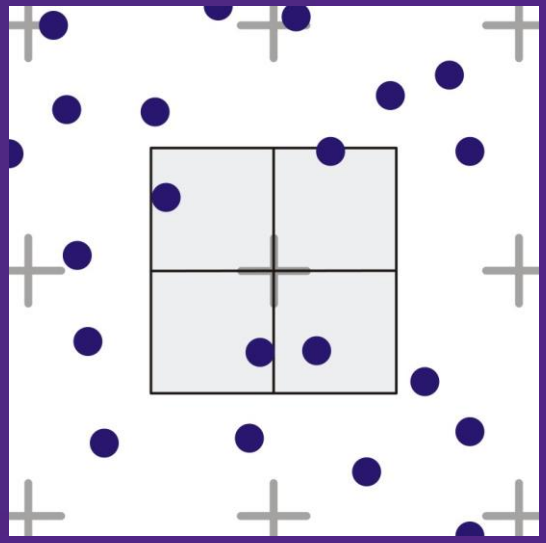
Antialiasing

- 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 in the following slide:

Antialiasing

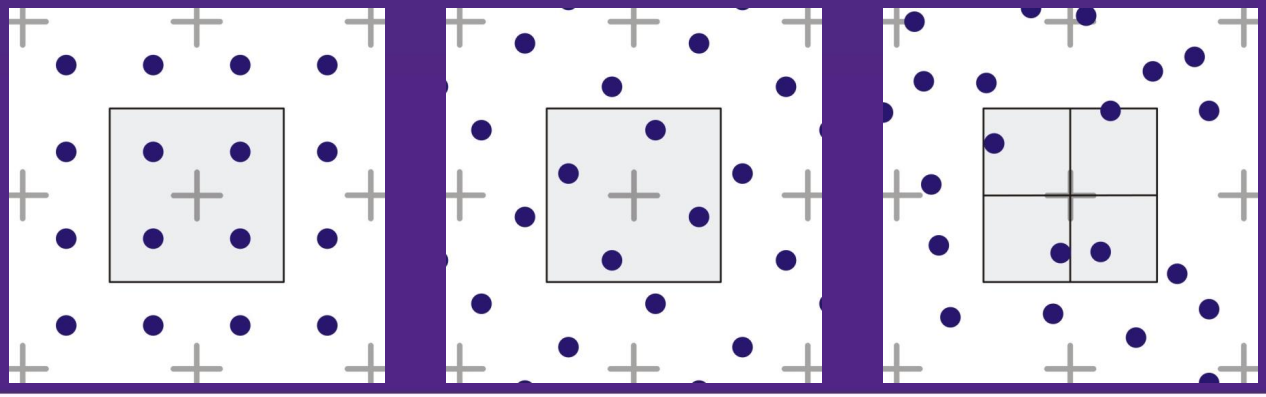




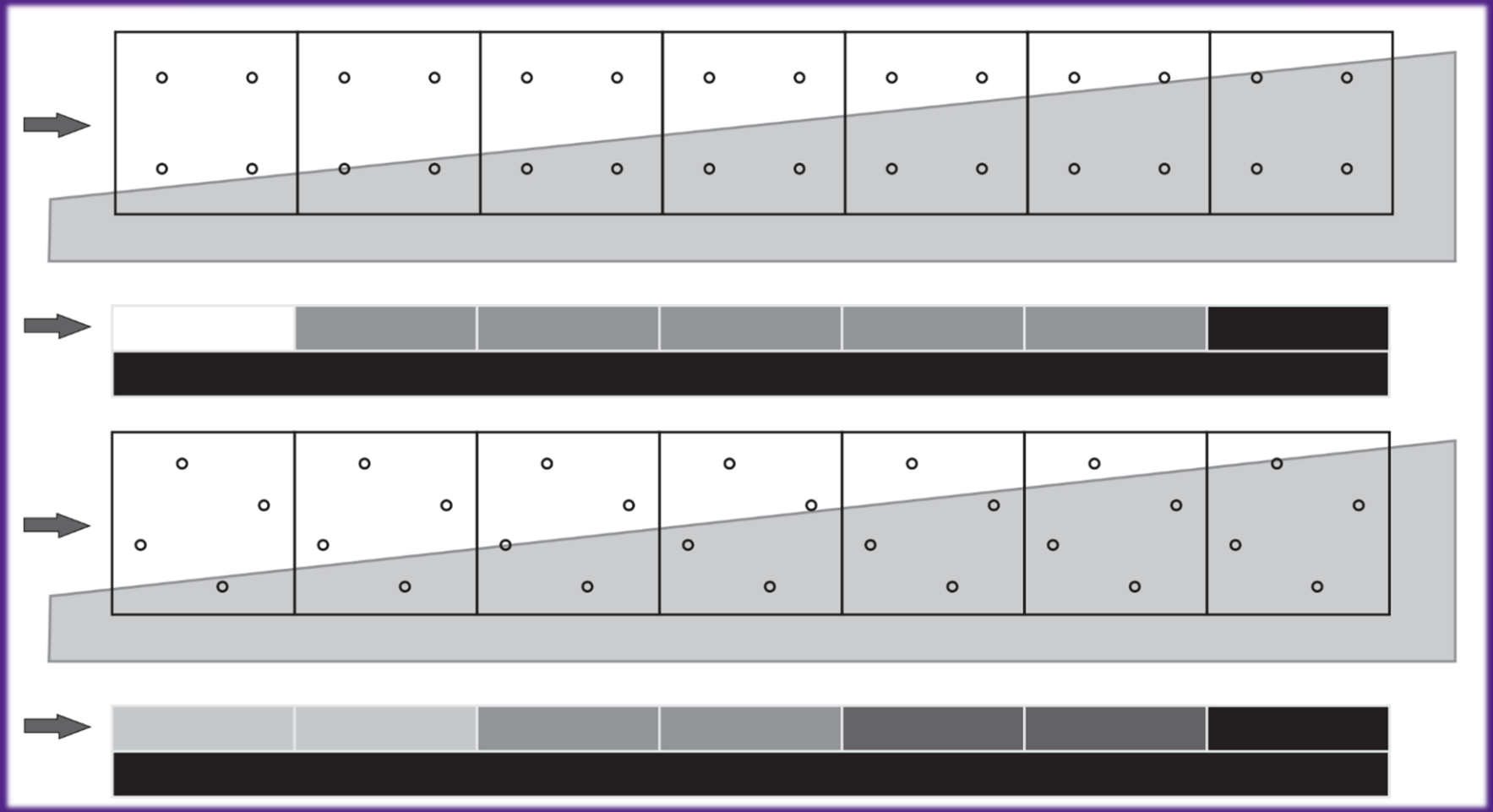


Antialiasing

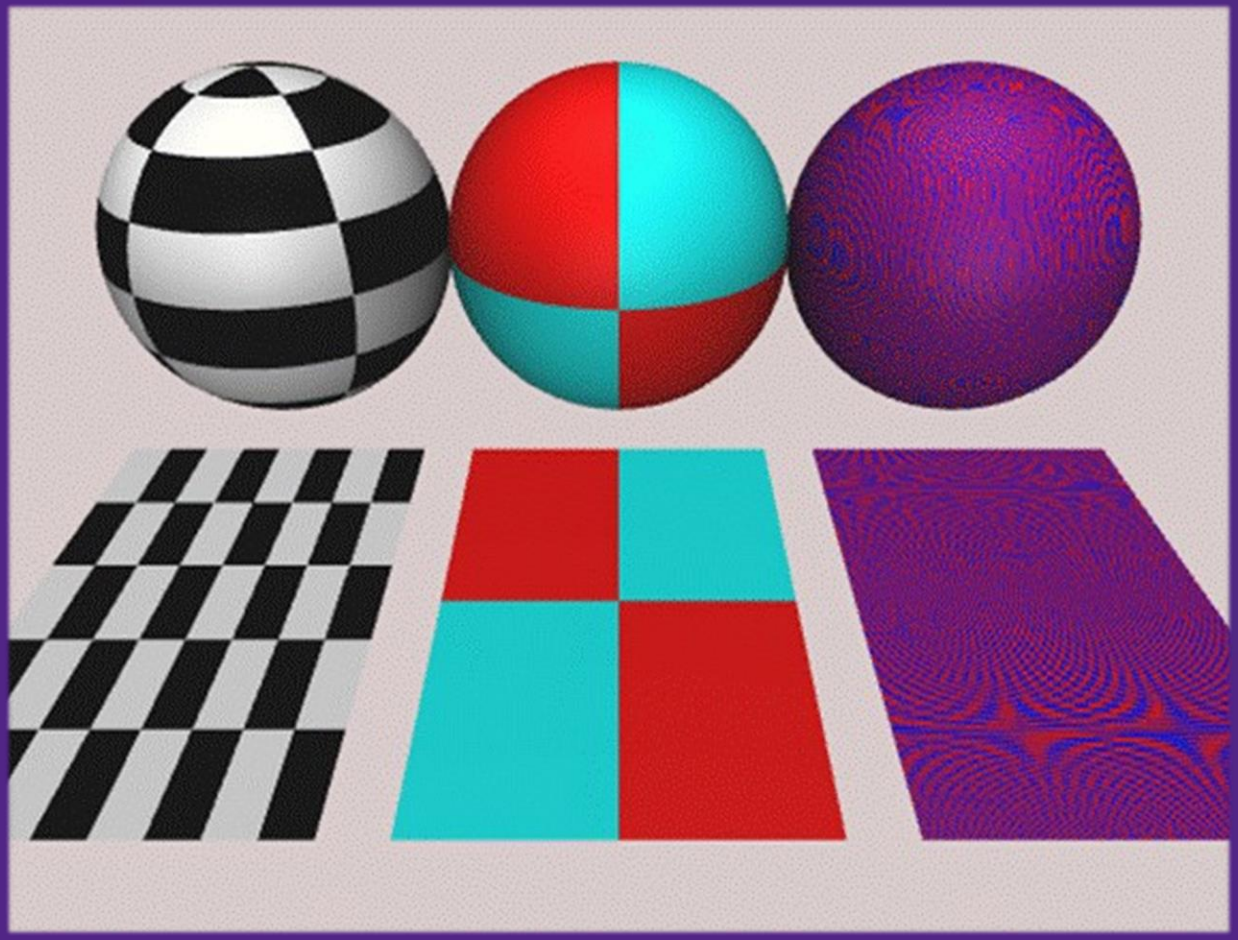
- While grid is the easiest to implement and very fast in practice, it produces less than stellar results.
- Visually, our eyes are quite sensitive to “nearly horizontal” and “nearly vertical” lines.
- The regular sample points of the grid pattern are bad at handling such nearly horizontal /vertical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



Antialiasing

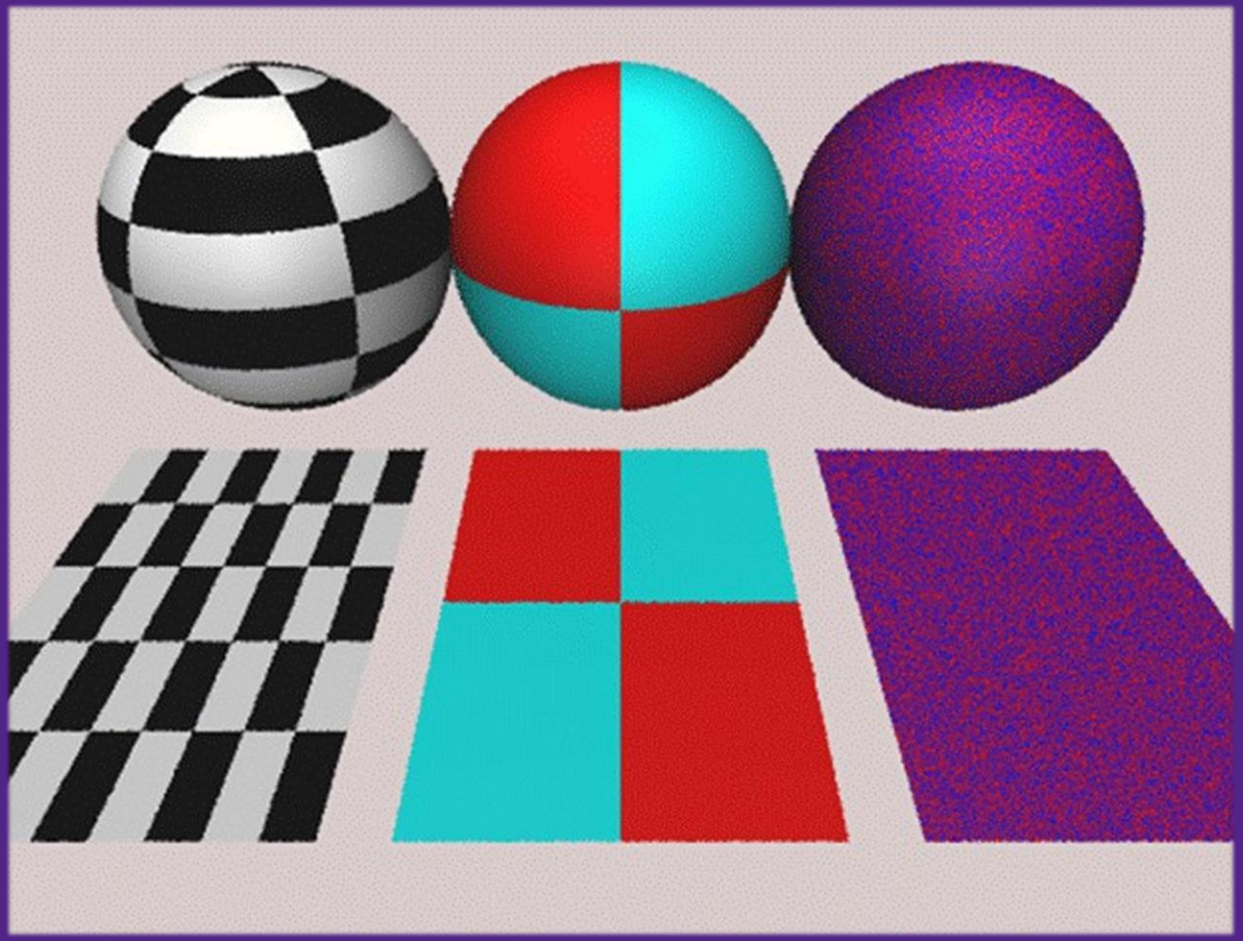


Antialiasing



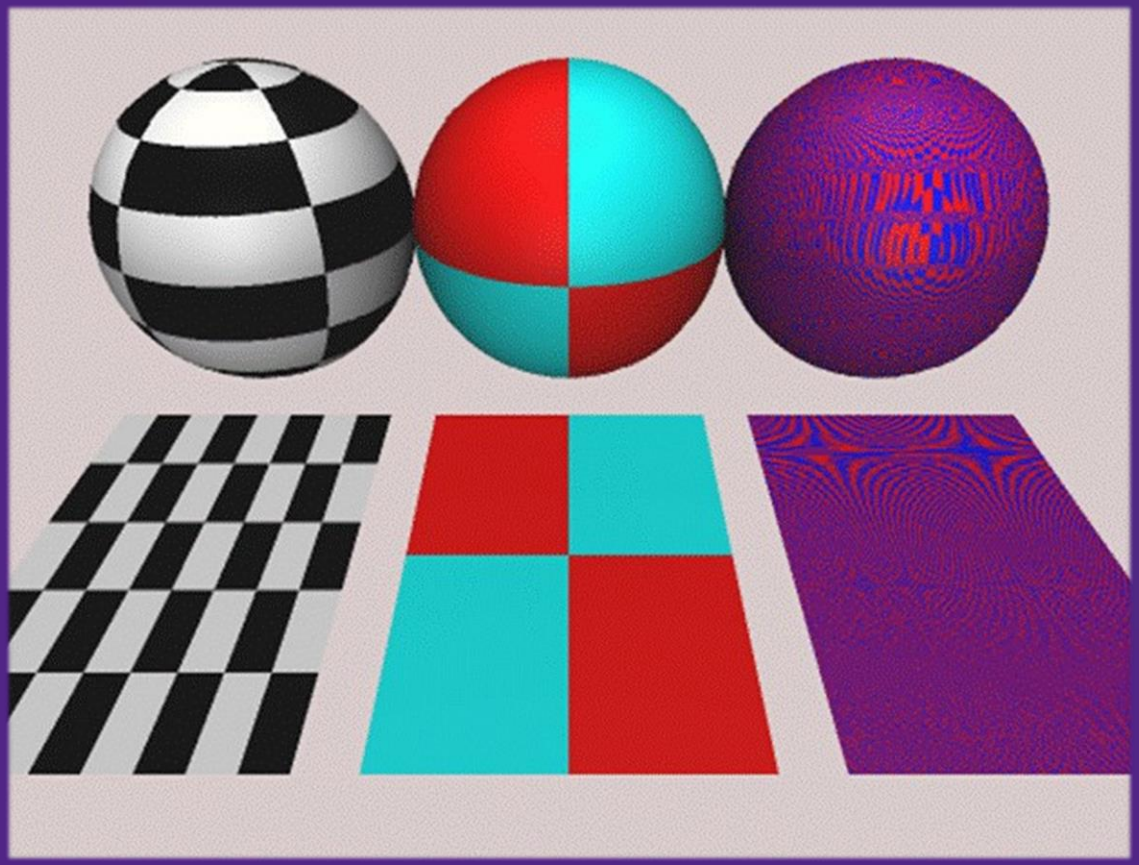
- Grid pattern

Antialiasing



- RGSS pattern

Antialiasing



- Jitter pattern