Run Length Encoding

By the end of this worksheet you should:

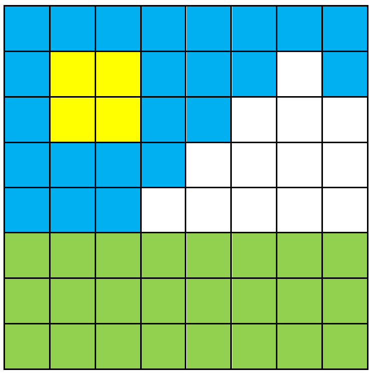
- Explain how data can be compressed using run length encoding (RLE)
- Represent data in RLE frequency/data pairs

Another type of lossless compression is Run Length Encoding or RLE.

This method is mainly used for reducing the size of image files, though there are now better, more modern options available.

Think back to when we created images; we saw that we could represent each pixel using binary patterns.

Well, imagine that we record each pixel, not by its bit pattern but by its colour:



So, we have the colours blue (B), yellow (Y), white (W) and green (G). Let's replace the colours with their codes:

B	B	B	B	B	B	B	B
B	Y	Y	B	B	B	W	B
B	Y	Y	B	B	W	W	W
B	B	B	B	W	W	W	W
B	B	B	W	W	W	W	W
G	G	G	G	G	G	G	G
G	G	G	G	G	G	G	G
G	G	G	G	G	G	G	G

Now we have our colour codes we can show it as a list of colours, starting from the top left (computers always start from top left; weird eh?!):

BBBBBBBBBYYBBBWBBYYBBWWWBBBBWWWWBB
BWWWWWGGGGGGGGGGGGGGGGGGGGGGGG

We can see that there are 4 colours, so we need 2 bits per pixel minimum, and we have 64 pixels in total; so we have 2 x 64 bits or 128 bits in total.

Our image would take up an estimated 128 bits storage space in total.

Okay, so how would RLE reduce this file without reducing the quality of the image?

Count each of the blocks of colour and you get:

9B 2Y 3B 1W 2B 2Y 2B 3W 4B 5W 3B 5W 24G

Largest number is 24, so we need 5 bits to store the value and 2 bits to store the colour. We have 13 colour groups (9B is 1 group, 2Y is another group, etc.).

So, we know we need 7 bits per group (5 for quantity, 2 for colour) and we have 13 groups; so this file will take up 7 bits x 13 or 91 bits.

Our new file uses only 91 bits, compared to 128 bits for the original file. We have saved 37 bits in total.

Now, that's neat but we know that computers work with bits but cannot work with a smaller number than 8 bits (a byte), so what would the real saving be?

Remember that each pixel *could* use only 2 bits?

Well, it couldn't; the smallest it could use would be a byte (8 bits). We have 64 pixels, each using 1 byte and so we have 64 bytes or 512 bits (64 x 8 bits).

RLE would use a byte for the value and a byte for the colour.

There are 13 groups, each consisting of 2 bytes; that makes 13 x 2 or 26 bytes, which is 208 bits (26 x 8).

Now that IS an incredible saving, without losing any detail at all.

We have saved 304 bits (512 – 208), or 38 bytes (64 – 26).

Imagine what we could save on a large photograph!

What might a GCSE exam paper ask you to do?

Imagine a picture that had one section made up of black (B) and white (W) pixels:

BBBBBBBBBBWWWWWW

- How would this be encoded using RLE?
- What might be the potential file size saving?

To answer this question, first show how RLE would reduce the colours into groups.

You would then state that the computer would normally have allocated 1 byte for each pixel, whilst it would allocate 1 byte to each value and 1 byte to each pixel.

You would then show how this affects the file size:

First I would create groups for each colour: 10B6W

The computer would normally use 1 byte for each pixel. There are 18 pixels so the original image would use 18 bytes in total.

RLE would give 1 byte to each value and 1 byte to each pixel colour. There are 2 values and 2 pixel colours, so the RLE file would use 4 bytes in total.

So, the original file used 18 bytes and the RLE file used only 4 bytes.

That is a saving of 14 bytes or 112 bits (14 x 8).

This answer shows you know

- what RLE is
- How an image is normally stored
- how RLE would reduce the file
- what a byte is
- how many bits in a byte

That'd make an excellent result!

Computer Graphics
CS 4731 Lecture 25
Polygon Filling & Antialiasing

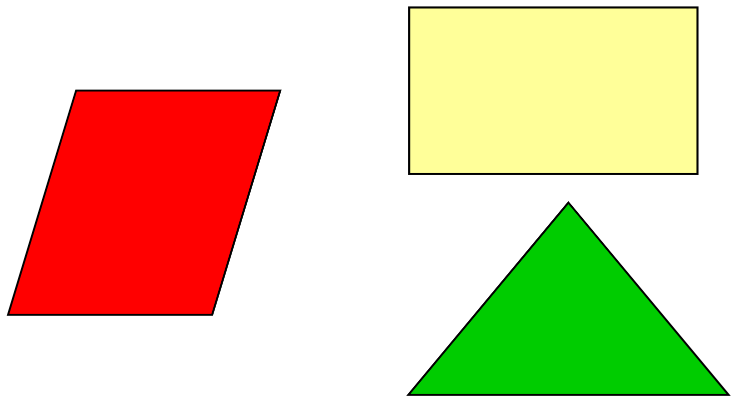
Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Defining and Filling Regions of Pixels

- Methods of defining region
  - **Pixel-defined:** specifies pixels in color or geometric range
  - **Symbolic:** provides property pixels in region must have
  - Examples of symbolic:
    - Closeness to some pixel
    - Within circle of radius *R*
    - Within a specified polygon



Pixel-Defined Regions

- **Definition:** Region R is the set of all pixels having color C that are connected to a given pixel S
- **4-adjacent:** pixels that lie next to each other horizontally or vertically, NOT diagonally
- **8-adjacent:** pixels that lie next to each other horizontally, vertically OR diagonally
- **4-connected:** if there is unbroken path of 4-adjacent pixels connecting them
- **8-connected:** unbroken path of 8-adjacent pixels connecting them

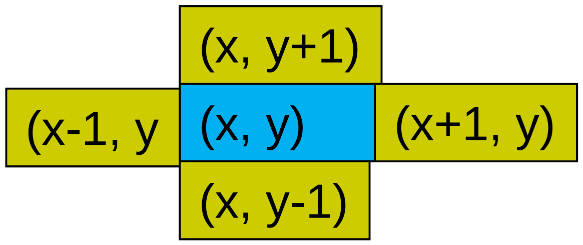
Recursive Flood-Fill Algorithm

- Recursive algorithm
- Starts from initial pixel of color, `intColor`
- Recursively set 4-connected neighbors to `newColor`
- **Flood-Fill**: floods region with `newColor`
- **Basic idea:**
  - start at "seed" pixel (x, y)
  - If (x, y) has color **intColor**, change it to **newColor**
  - Do same recursively for all 4 neighbors

Recursive Flood-Fill Algorithm

- **Note:** getPixel(x,y) used to interrogate pixel color at (x, y)

```
void floodFill(short x, short y, short intColor)
{
   if(getPixel(x, y) == intColor)
   {
        setPixel(x, y);
        floodFill(x – 1, y, intColor); // left pixel
        floodFill(x + 1, y, intColor); // right pixel
        floodFill(x, y + 1, intColor); // down pixel
        floodFill(x, y – 1, intColor); // up pixel
   }
}
```

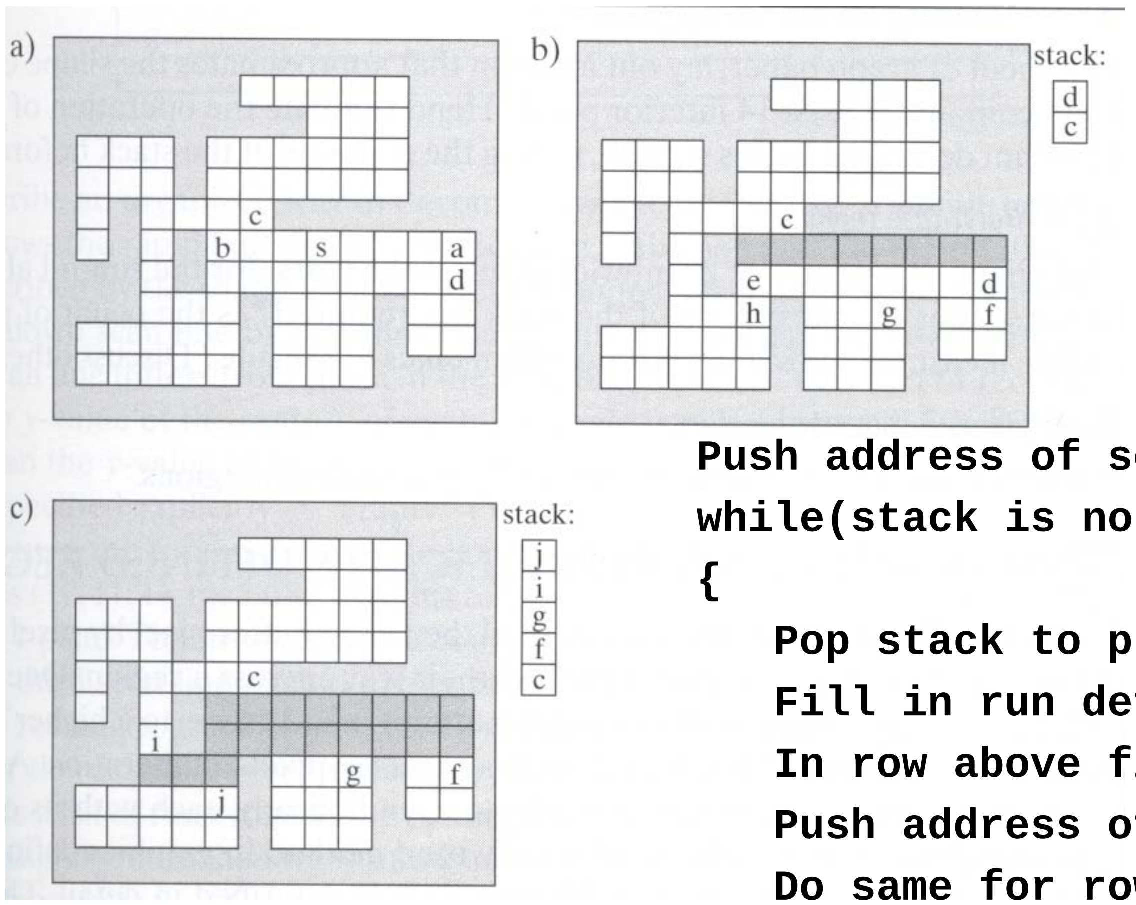


Recursive Flood-Fill Algorithm

- Recursive flood-fill is blind
- Some pixels retested several times
- **Region coherence** is likelihood that an interior pixel mostly likely adjacent to another interior pixel
- **Coherence** can be used to improve algorithm performance
- **A run:** group of adjacent pixels lying on same scanline
- Fill runs(adjacent, on same scan line) of pixels

Region Filling Using Coherence

- Example: start at s, initial seed



Pseudocode:

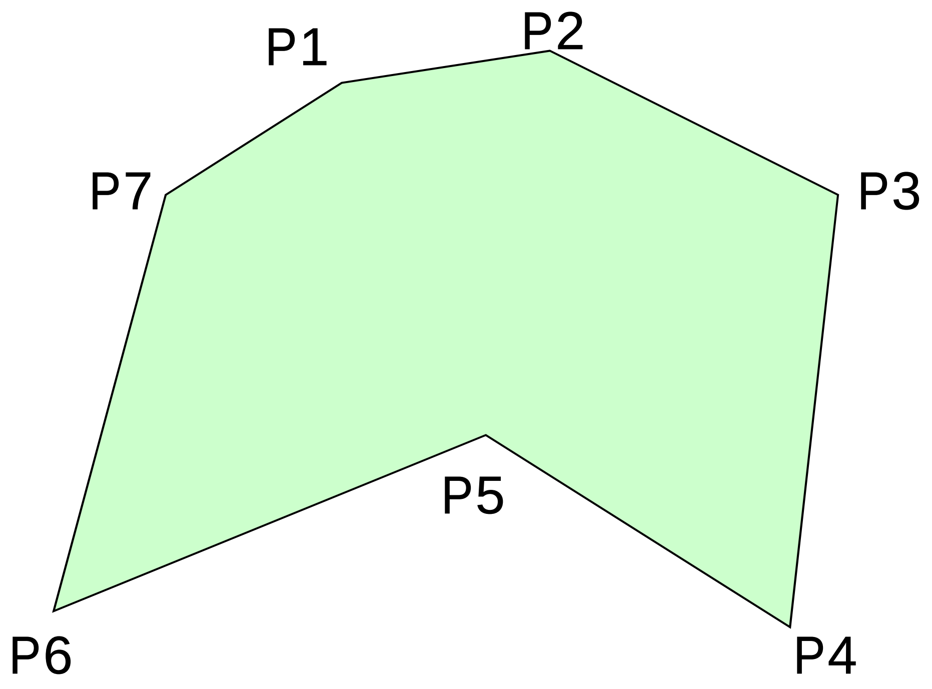
```
Push address of seed pixel onto stack
while(stack is not empty)
{
    Pop stack to provide next seed
    Fill in run defined by seed
    In row above find reachable interior runs
    Push address of their rightmost pixels
    Do same for row below current run
}
```

Note: algorithm most efficient if there is **span coherence** (pixels on scanline have same value) and **scan-line coherence** (consecutive scanlines similar)

Filling Polygon-Defined Regions

- **Problem:** Region defined polygon with vertices

  Pi = (Xi, Yi), for i = 1…N, specifying sequence of P's vertices



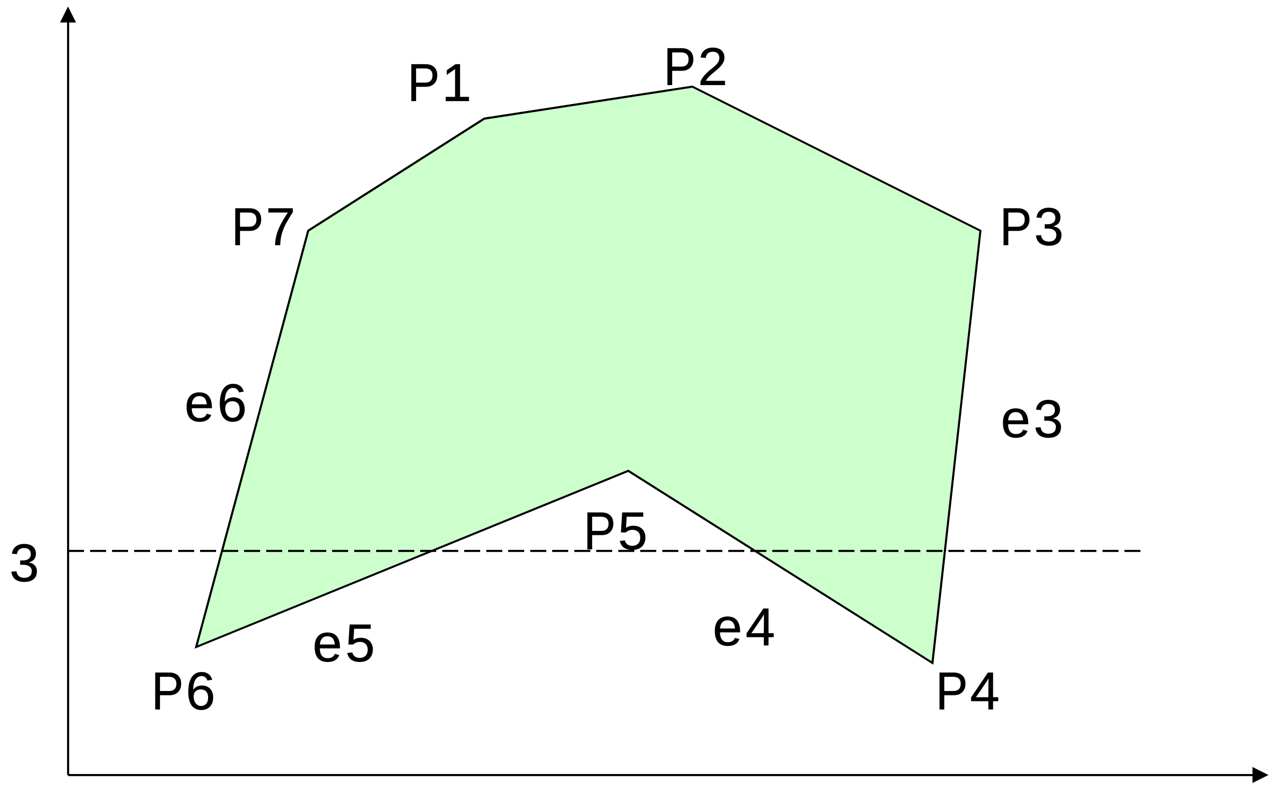
Filling Polygon-Defined Regions

- **Solution:** Progress through frame buffer scan line by scan line, filling in appropriate portions of each line
- Filled portions defined by intersection of scan line and polygon edges
- Runs lying between edges inside P are filled
- **Pseudocode:**

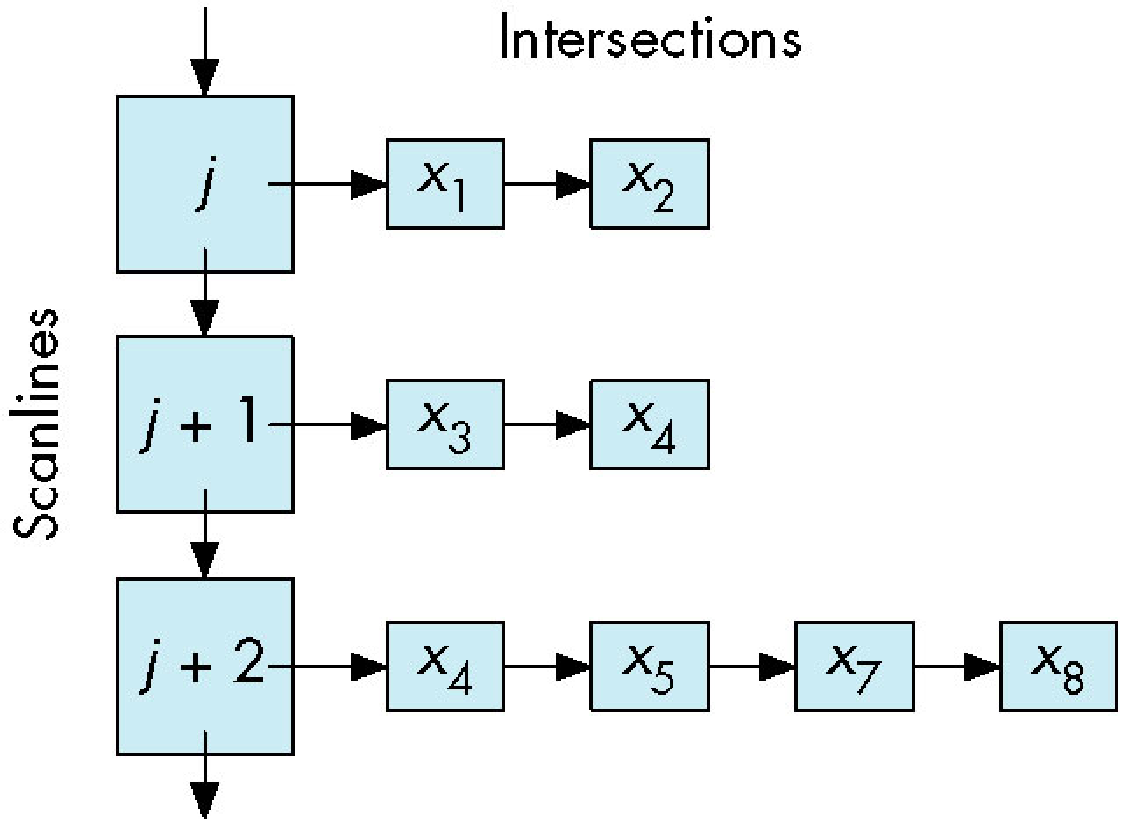
```
for(each scan Line L)
{
   Find intersections of L with all edges of P
   Sort the intersections by increasing x-value
   Fill pixel runs between all pairs of intersections
}
```

Filling Polygon-Defined Regions

- **Example:** scan line y = 3 intersects 4 edges e3, e4, e5, e6
- Sort x values of intersections and fill runs in pairs
- **Note:** at each intersection, inside-outside (parity), or vice versa

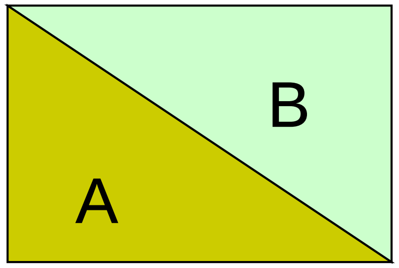


Data Structure



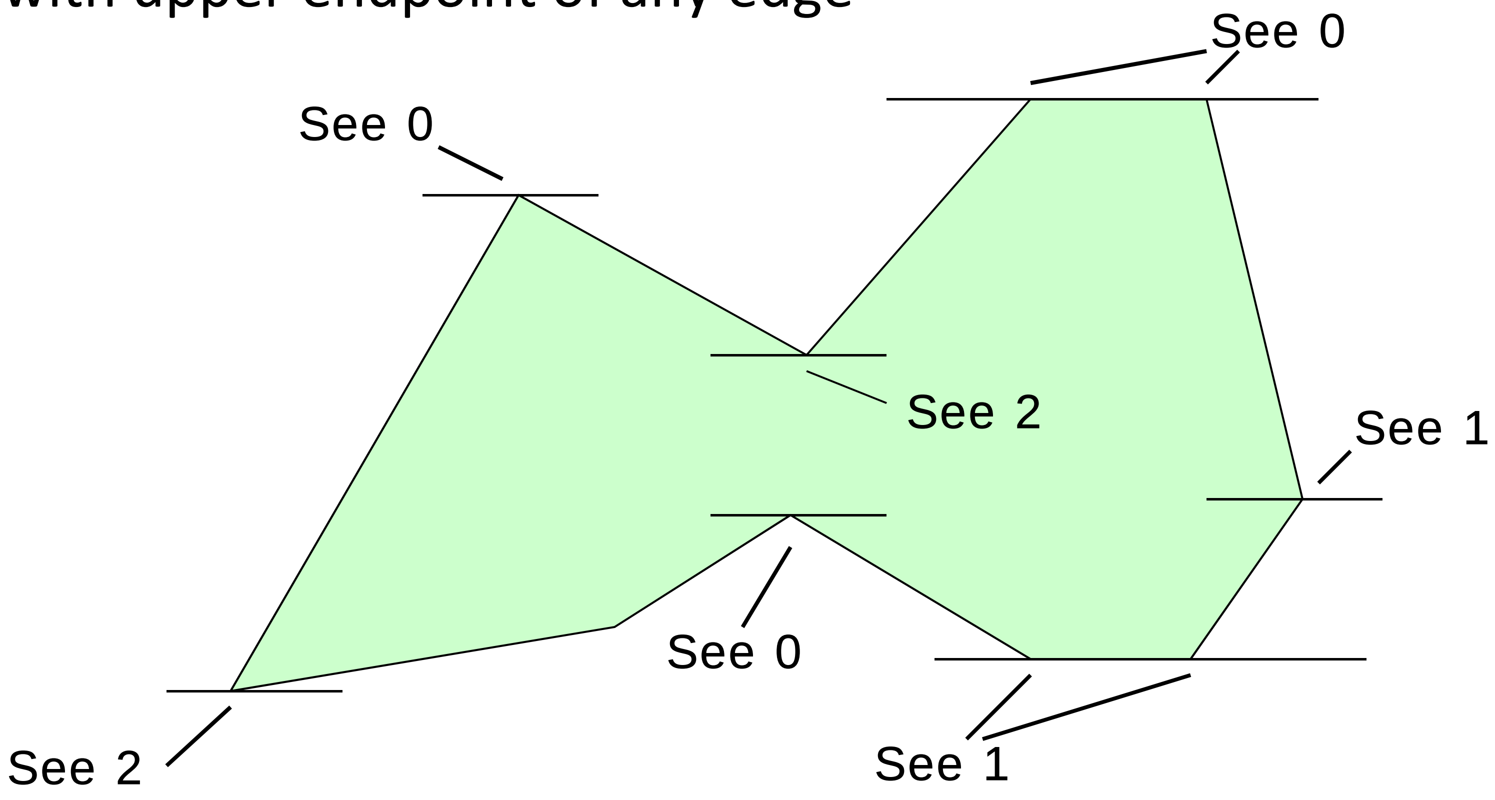
Filling Polygon-Defined Regions

- **Problem:** What if two polygons A, B share an edge?
- Algorithm behavior could result in:
  - setting edge first in one color and the another
  - Drawing edge twice too bright
- **Make Rule:** when two polygons share edge, each polygon owns its left and bottom edges
- E.g. below draw shared edge with color of polygon **B**



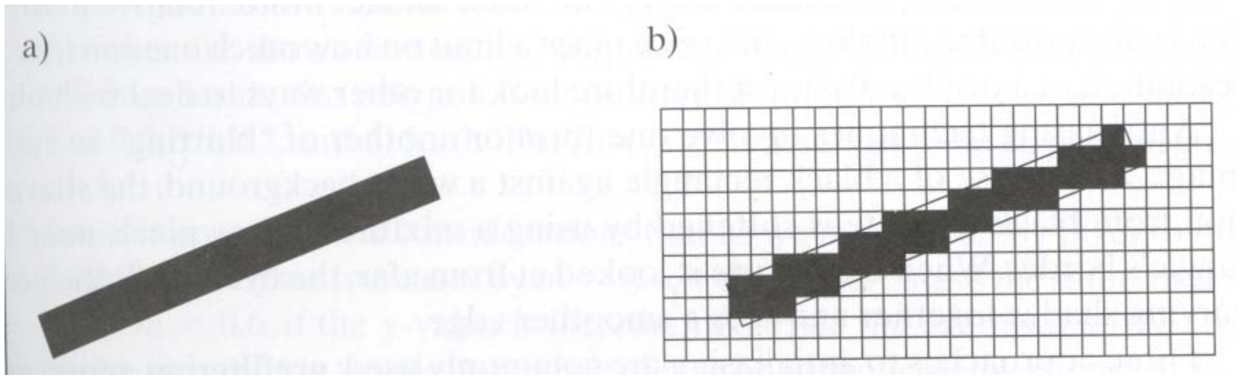
Filling Polygon-Defined Regions

- **Problem:** How to handle cases where scan line intersects with polygon endpoints to avoid wrong parity?
- **Solution:** Discard intersections with horizontal edges and with upper endpoint of any edge



Antialiasing

- Raster displays have pixels as rectangles
- Aliasing: Discrete nature of pixels introduces "jaggies"



Antialiasing

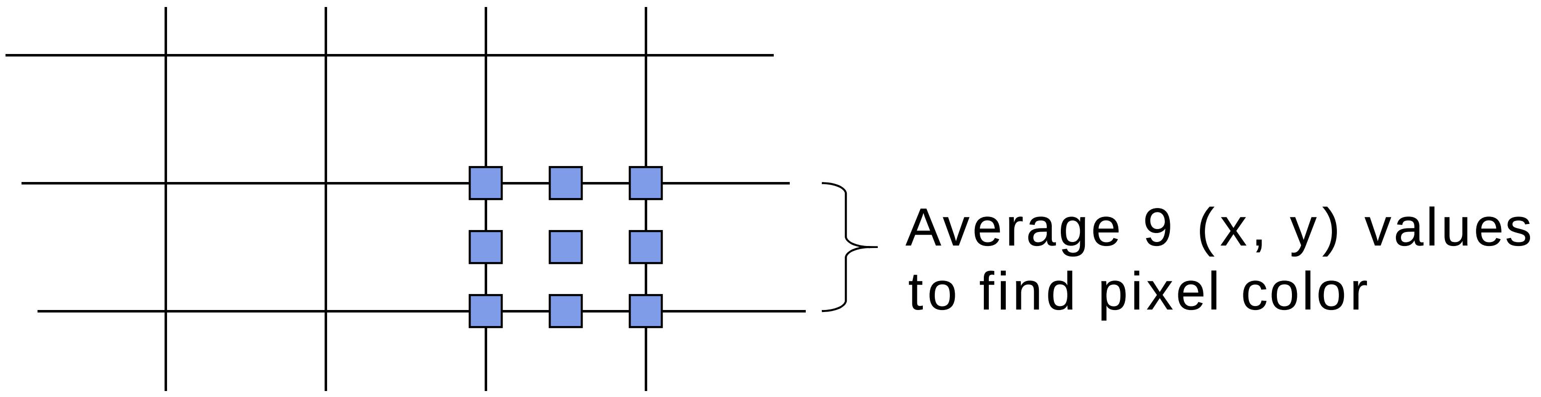
- Aliasing effects:
  - Distant objects may disappear entirely
  - Objects can blink on and off in animations
- Antialiasing techniques involve some form of blurring to reduce contrast, smoothen image
- Three antialiasing techniques:
  - Prefiltering
  - Postfiltering
  - Supersampling

Prefiltering

- Basic idea:
  - compute area of polygon coverage
  - use proportional intensity value
- Example: if polygon covers ¼ of the pixel
  - Pixel color = ¼ polygon color + ¾ adjacent region color
- Cons: computing polygon coverage can be time consuming

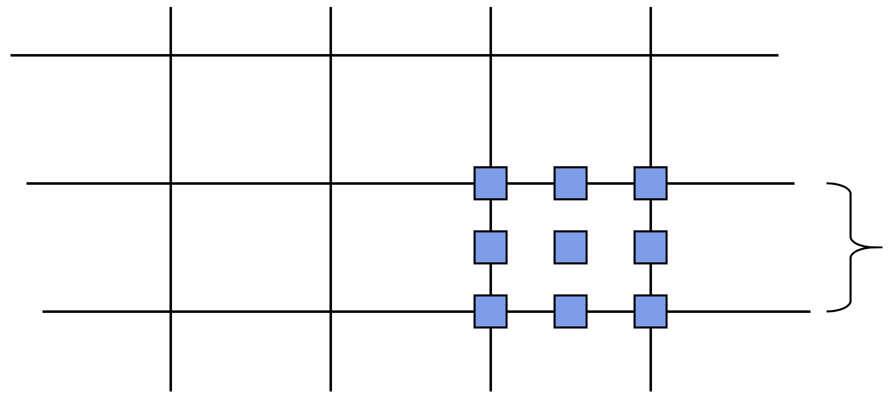
Supersampling

- Assumes we can compute color of any location (x,y) on screen
- Increase frequency of sampling
- Instead of (x,y) samples in increments of 1
- Sample (x,y) in fractional (e.g. ½) increments, average samples
- Example: Double sampling = increments of ½ = 9 color values averaged for each pixel



Postfiltering

- Supersampling uses average
- Gives all samples equal importance
- Post-filtering: use weighting (different levels of importance)
- Compute pixel value as weighted average
- Samples close to pixel center given more weight



Sample weighting

1 / 16	1 / 16	1 / 16
1 / 16	1 / 2	1 / 16
1 / 16	1 / 16	1 / 16

Antialiasing in OpenGL

- Many alternatives
- Simplest: accumulation buffer
- **Accumulation buffer:** extra storage, similar to frame buffer
- Samples are accumulated
- When all slightly perturbed samples are done, copy results to frame buffer and draw

Antialiasing in OpenGL

- First initialize:
  - `glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH);`
- Zero out accumulation buffer
  - `glClear(GLUT_ACCUM_BUFFER_BIT);`
- Add samples to accumulation buffer using
  - `glAccum( )`

Antialiasing in OpenGL

- Sample code
- jitter[] stores randomized slight displacements of camera,
- factor, f controls amount of overall sliding

```
glClear(GL_ACCUM_BUFFER_BIT);
for(int i=0;i < 8; i++)
{
   cam.slide(f*jitter[i], f*jitter[i].y, 0);
   display( );
   glAccum(GL_ACCUM, 1/8.0);
}
glAccum(GL_RETURN, 1.0);
```

jitter.h

-0.3348, 0.4353

0.2864, -0.3934

……

References

- Hill and Kelley, chapter 11
- Angel and Shreiner, Interactive Computer Graphics, 6th edition

Run-Length Encoding (RLE)

Run-length encoding (RLE) is one of the simplest data compression methods. Consider the example in which we have represented an MxN image whose top half s totally white, and bottom half is totally black. That example was a primitive attempt to encode the image using RLE. The principle of RLE is to exploit the repeating values in a source. The algorithm counts the consecutive repetition amount of a symbol and uses that value to represent the **run**. This simple principle works best on certain source types in which repeated data values are significant. Black-White document images, cartoon images, etc. are quite suitable for RLE.

Actually, RLE may be used on any kind of source regardless of its content, but its compression efficiency changes significantly depending on the above types of data are used or not.

As another application suitable for RLE, we can mention text files which contains multiple spaces for indention and formatting paragraphs, tables and charts.

Principle :
As indicated above, basic RLE principle is that the run of characters is replaced with the number of the same characters and a single character. Examples may be helpful to understand it better.
Ex. 8:
Consider a text source: R T A A A A S D E E E E E
The RLE representation is: R T *4A S D *5E
This example also shows how to distinguish whether a symbol corresponds to the daa value or its repetition count (called run). Each repeating bunch of characters is replaced with three symbols: **an indicator (*), number of characters**, and the **character itself**. We need the indication of he redundant character * to separate between the encoding of a repeating cluster and a single character. In the above example, if there is no repetition around a character, it is encoded as itself.

In the above example, it is important to realize that the encoding process is effective only if there are sequences of 4 or more repeating characters because three characters are used to conduct RLE. For example, coding two repeating characters would lead to expansion and coding three repeating characters wouldn't cause compression or expansion since we represent a repetitive cluster with at least three symbols.

The decoding process is easy: If there aren't control characters (*) the coded symbol just corresponds to the original symbol, and if control character occurred then it must be replaced with characters in a defined number of times. It can be noticed that the process of decoding control characters don't lead to any special procedures.

The RLE symbolism and details are not unique. There are many different run-length encoding schemes. The above example has just been used to demonstrate the basic principle of RLE encoding. In a particular case the implementation of RLE depends on what type of data is being compressed.

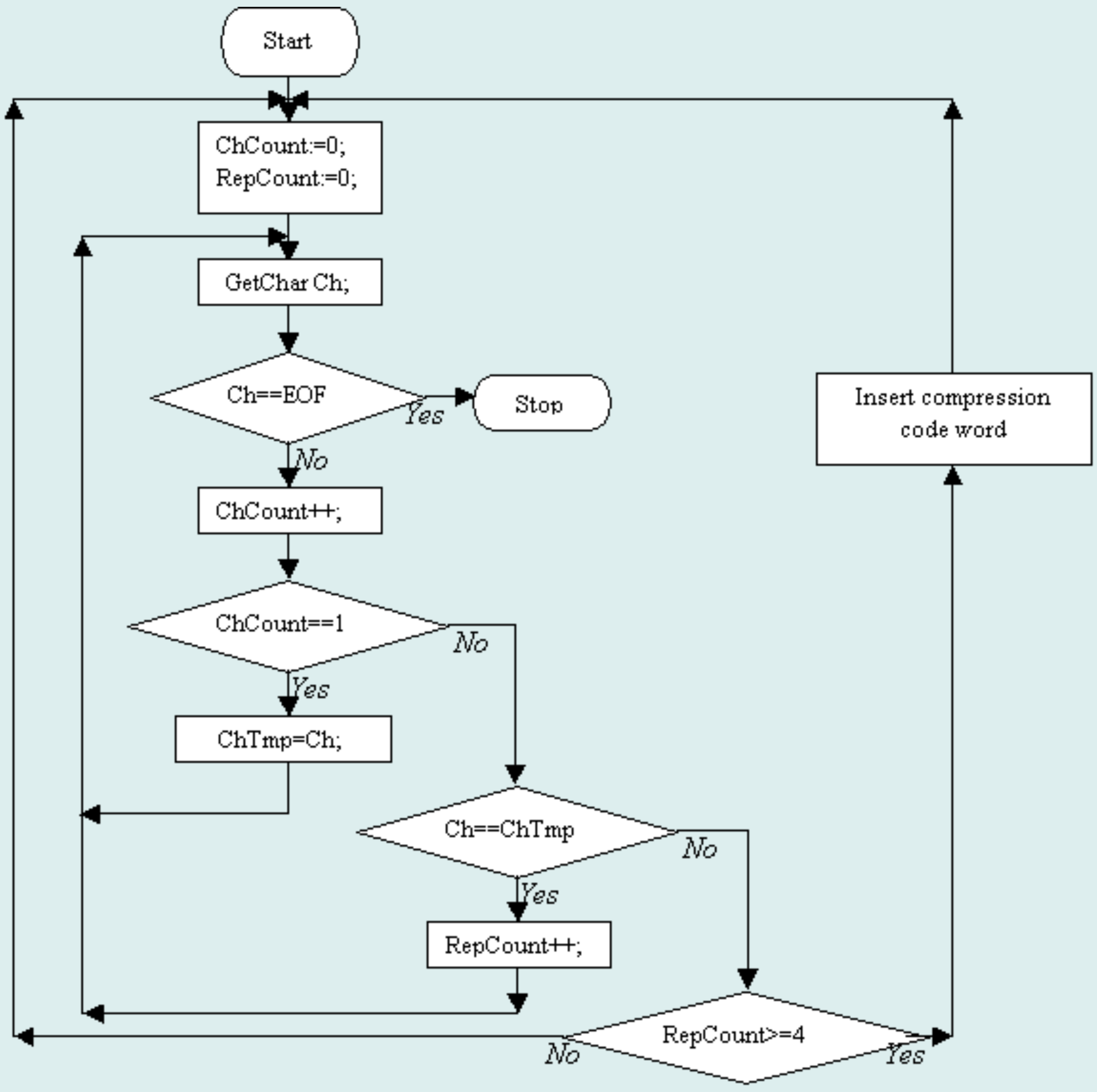
Nevertheless the encoder applies a three symbol encoding strategy to represent a repetitive cluster:

CTRL	COUNT	CHAR

Here, the following terminology is defined:

- CTRL - control character which is used to indicate compression
- COUNT- number of counted characters in stream of the same characters
- CHAR - repeating characters

The chart below illustrates a typical RLE encoder for text files. Similar algorithms and charts may be adopted for image, audio, etc.



RLE algorithms are practically used in various image compression techniques like the well known BMP, PCX, TIFF, and is also used in PDF file format. Furthermore, RLE also exists as separate compression technique and there is also a file format called RLE (in various brands).

One of the companies that incorporates RLE is **CompuServe**. The file format is threefore CompuServe RLE, which was standardized in the 80's and it was aimed to compress for 1-bit (black and white) images.

Note: The standards are usually patented. They contain specific headers that indicate the technique and the name of the standard. For example, in CompuServe RLE, header part indicates or represents the Graphic Mode Control, which indicates that it is a CompuServe image. The standard initial header symbol sequence is : ASCII ESC (HEX 1B), ASCII G(HEX 47) and the third character is ASCII H (HEX 48) or M (HEX 4D). Third character represents resolution (out of two possible graphics modes : high resolution graphic mode (256 x 192 pixels) represented by <H> and medium resolution graphic mode (128 x 96 pixels) represented by <M>). Therefore, typically, if a file starts with <ESC><G><H> or <ESC><G><M>, then it is a CompuServe RLE image.

After header sequence, the application can identify that it is a CompuServe RLE image. Now the data sequence starts.

Basic data sequence consists of a pair of run length encoded ASCII characters. The first number represents number of the background (off) pixels and the second character is the number of foreground (on) pixels. Each number of a pair represents the count number of pixels plus 32 decimal, i.e. from each number 32 is substracted and that number represents how many next pixels will be turned on or turned off depending on what number of pair we observe. Usually the value 126 is used as the highest possible value, because of some limitations in some old computer terminals. This limitation leads us to the conclusion that in each byte we can denote repetition of at most 94 pixels (126 - 32). For example pair <D><'> (HEX: 44 27, DECIMAL 68 39) means next 68 (decimal) pixels are turned off and then 39 (decimal) pixels are turned on.

As you see, there is no control character (like '*') between the encoded symbols because there are only two distinct source levels: 0 and 1. For more general data types (color o gray-level images), CompuServe RLE is modified to incorporate the control chaacter.

Finally, the file formats must also define where the input data ends. The ending sequence for RLE standard consists of three characters <ESC><G><N>.

Ex. 9: A typical CompuServe RLE file (all numbers are in ASCII HEX format):

```
1B 47 48 7E 20 7E 20 7E 20 7E 20 ....
. . . . . . . . . . . .
. 41 36 . . . . . . . . .
. . . . . . . . . . . .
. . . 07 1B 47 4E
```

1B 47 48 - is header <ESC><G><H> and represents high resolution, first data sequence pair 20hex, 7Ehex means that first 94dec pixels are all turned on, the second data sequence is the same so second 94d pixels are also turned on (the first 188d pixels are turned on so far), and so on. Then somewhere in the file pairs 41h 36h occurs which means that next 33d pixels are turned off and after that 22d pixels are turned off, etc. Last four character are the ending sequence which was described above.

You may want to experiment with some real image data. Click here to download samples of CompuServe RLE images

Another standard that uses RLE is **MS Windows standard for RLE** file format. MS Windows standard for RLE have the same file format (in terms of headers, etc.) as the well-known BMP file format, but it's RLE format is defined only for 4-bit and 8-bit colour images.

4bit RLE: Compression sequence consists of two bytes, first byte (if not zero) determines number of pixels which will be drawn on the screen. The second byte specifies two colors, high-order 4 bits (upper 4 bits) specifies the first color, low-order 4bits specifies the second color. This means that 1st, 3rd and other odd pixels will be in drawn in the color specified by high-order bits, while 2nd, 4th and other even pixels will be in drawn in the color specified by low-order bits. If first byte is zero then the second byte specifies escape code which describes some specific behavior such as:

- 0 : End-of-line
- 1 : End-of-Rle(Bitmap)
- 2 : Following two bytes defines offset in x and y direction (x is right,y is up). The skipped pixels get color zero.
- >=3 : When expanding (decoding), the following >=3 nibbles (nibble means 4bits) are just copied from compressed file, file/memory pointer must be on 16bit boundary so adequate number of zeros follows

Ex. 10 : Example for 4bit RLE:

Compressed data	Expanded data
06 52	5 2 5 2 5 2
08 1B	1 B 1 B 1 B 1 B
00 06 83 14 34	8 3 1 4 3 4
00 02 09 06	Move 9 positions right and 6 up
00 00	End-of -line
04 22	2 2 2 2
00 01	End-of-RLE(Bitmap)

8bit RLE: Compressed sequence is also formed from 2 bytes (like the 4 bit RLE). the first byte (if not zero) is a number of consecutive pixels which are in color specified by the second byte. Same as in 4bit RLE if the first byte is zero the second byte defines escape code, escape codes 0, 1, 2, have same meaning as described in 4bit RLE.

Ex. 11: Examples for 8bit RLE

Compressed data	Expanded data
06 52	

	52 52 52 52 52 52
08 1B	1B 1B 1B 1B 1B 1B 1B 1B
00 03 83 14 34	83 14 34
00 02 09 06	Move 9 positions right and 6 up
00 00	End-of -line
04 2A	2A 2A 2A 2A
00 01	End-of-RLE(Bitmap)

Detailed description about MS Windows BMP file format (RLE is special case of BMP) is available here.

Other RLE file formats:

RLE scheme which will be described here is being used in PDF and TIFF file format. In this case, RLE encoded data consists of compression sequences. One compression sequence starts with number n (byte), this byte may be followed by 1 to 128 bytes (so these 2 to 129 bytes form one compression sequence). If n is between 0 and 127, then following n+1 (1 to 128) bytes are just copied during decompression. If n is between 129 and 255, then the byte value which follows n is copied 256-(n-1) (which is between 2 and 128) times in the decompressed file. If 128 occurs then it indicates the end of compressed data.

This scheme is similar to the PackBits encoding standard which is used in Macintosh systes.

Ex. 12 : For the above description you can consider the following example:

Compressed data-hex format	Decompressed data-hex format
07 A4 56 C9 90 E5 F1 DB 32	A4 56 C9 E5 F1 DB 32
02 23 A1 56	23 A1 56
FE 12	12 12 12
FC 6C	6C 6C 6C 6C 6C

Please check the following resources for further detailed descriptions on RLE:

- Detailed description of Utah RLE
- Another useful link about Utah RLE - file format description
- DjVu compression technology - RLE file format description- specified by AT&T
- Portable Document Format Reference Manual Version 1.3 November 16, 1999
- Unofficial CompuServe RLE specification
- Specification of MS Windows BMP

CS 543: Computer Graphics
Lecture 9 (Part II): Raster Graphics: Polygons & Antialiasing

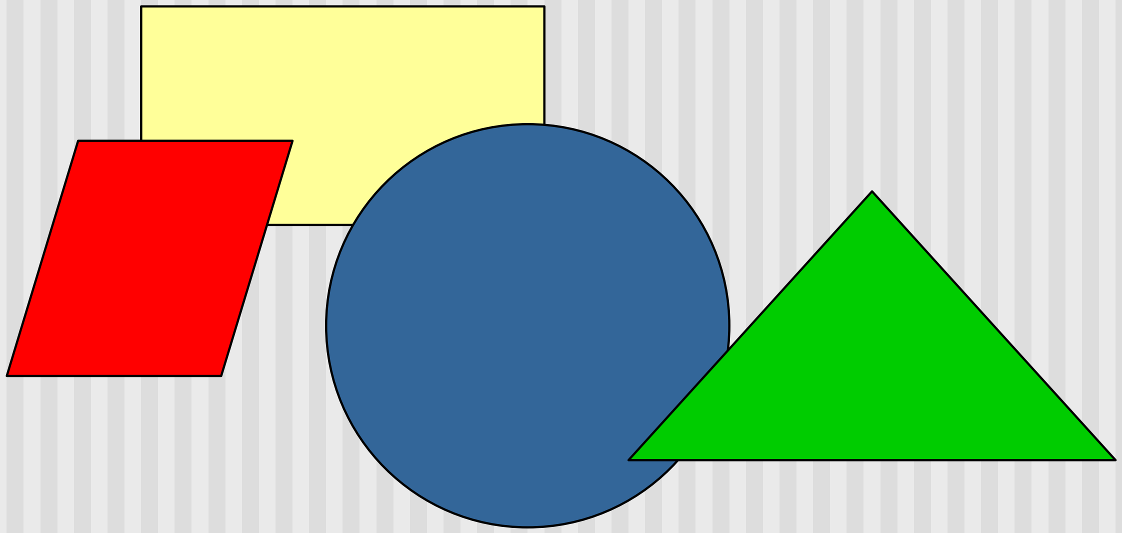
Emmanuel Agu

So Far…

- Raster graphics:
  - Line drawing algorithms (simple, Bresenham’s)
- Today:
  - Defining and filling regions
  - Polygon drawing and filling
  - Antialiasing

Defining and Filling Regions of Pixels

- First, understand how to define and fill any defined regions
- Next, how to fill regions bounded by a polygon



Defining and Filling Regions of Pixels

- Methods of defining region
    - Pixel-defined: specifies pixels in color or geometric range
    - Symbolic: provides property pixels in region must have
    - Examples of symbolic:
        - Closeness to some pixel
        - Within circle of radius *R*
        - Within a specified polygon

Pixel-Defined Regions

- **Definition:** Region R is the set of all pixels having color C that are connected to a given pixel S
- **4-adjacent:** pixels that lie next to each other horizontally or vertically, NOT diagonally
- **8-adjacent:** pixels that lie next to each other horizontally, vertically OR diagonally
- **4-connected:** if there is unbroken path of 4-adjacent pixels connecting them
- **8-connected:** unbroken path of 8-adjacent pixels connecting them

Recursive Flood-Fill Algorithm

- Recursive algorithm
- Starts from initial pixel of color, **`intColor`**
- Recursively set 4-connected neighbors to **`newColor`**
- **Flood-Fill**: floods region with **`newColor`**
- **Basic idea:**
  - start at “seed” pixel (x, y)
  - If (x, y) has color intColor, change it to newColor
  - Do same recursively for all 4 neighbors

Recursive Flood-Fill Algorithm

- **Note:** getPixel(x,y) used to interrogate pixel color at (x, y)

```
void floodFill(short x, short y, short intColor)
{
  if(getPixel(x, y) == intColor)
  {
      setPixel(x, y);
      floodFill(x – 1, y, intColor); // left pixel
      floodFill(x + 1, y, intColor); // right pixel
      floodFill(x, y + 1, intColor); // down pixel
      floodFill(x, y – 1, intColor); // fill up
  }
}
```

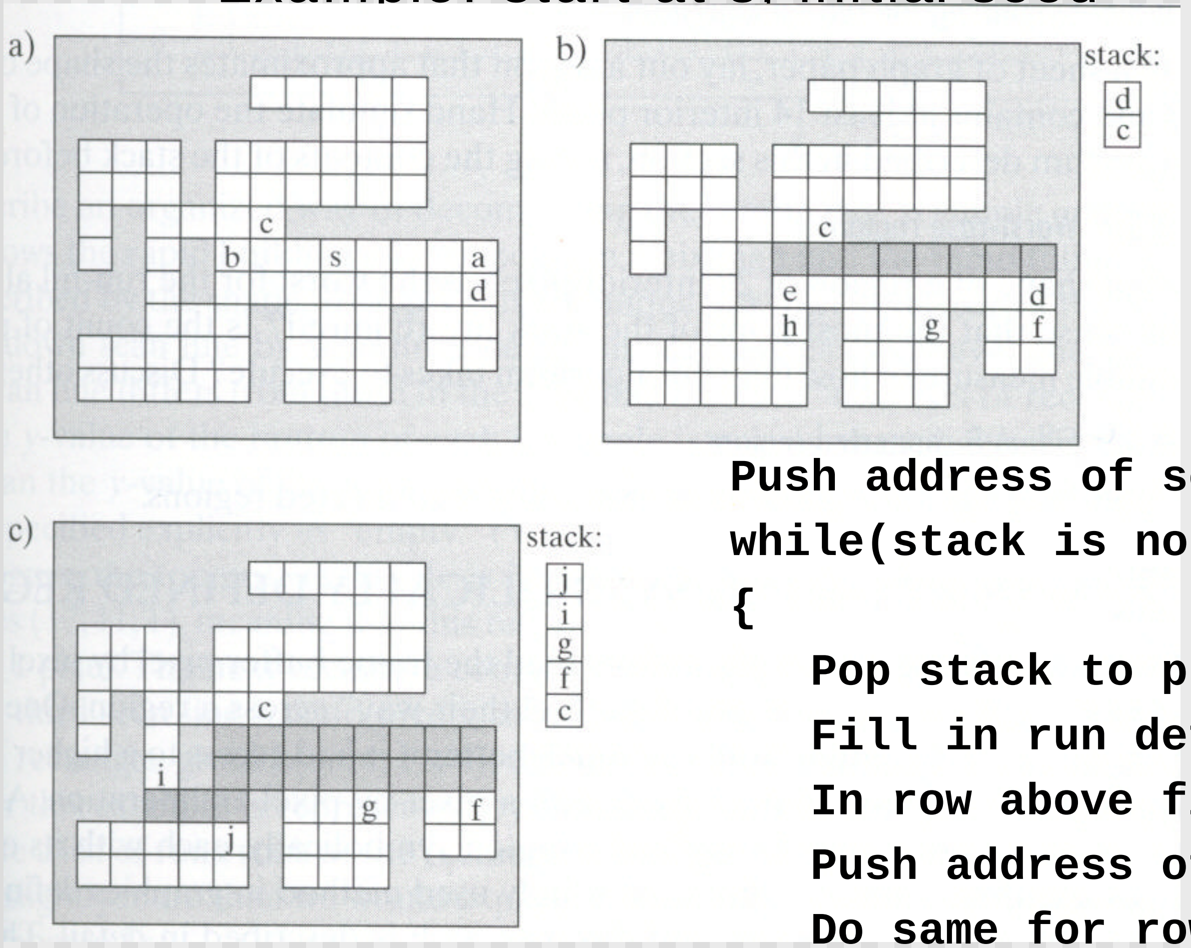
Recursive Flood-Fill Algorithm

- This version defines region using intColor
- Can also have version defining region by boundary
- Recursive flood-fill is somewhat blind and some pixels may be retested several times before algorithm terminates
- Region coherence is likelihood that an interior pixel mostly likely adjacent to another interior pixel
- Coherence can be used to improve algorithm performance
- A run is a group of adjacent pixels lying on same
- Exploit runs(adjacent, on same scan line) of pixels

Note: algorithm most efficient if there is **span coherence** (pixels on scanline have same value) and **scan-line coherence** (consecutive scanlines are similar)

Region Filling Using Coherence

- Example: start at s, initial seed

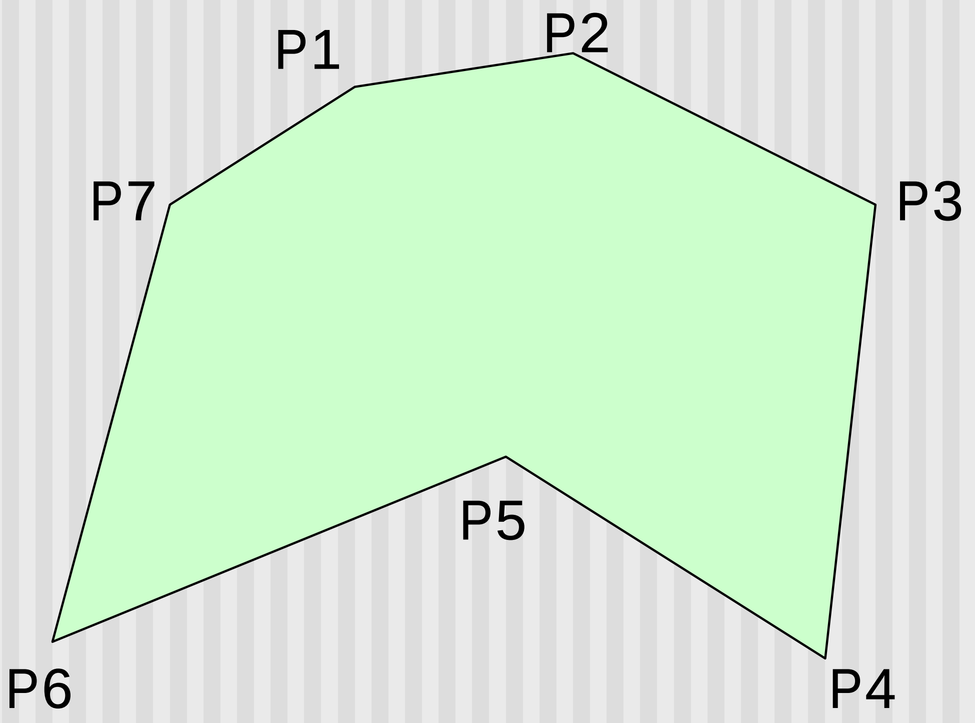


Pseudocode:

```
Push address of seed pixel onto stack
while(stack is not empty)
{
   Pop stack to provide next seed
   Fill in run defined by seed
   In row above find reachable interior runs
   Push address of their rightmost pixels
   Do same for row below current run
}
```

Filling Polygon-Defined Regions

- **Problem:** Region defined by Polygon P with vertices Pi = (Xi, Yi), for i – 1…N, specifying sequence of P's vertices



Filling Polygon-Defined Regions

- **Solution:** Progress through frame buffer scan line by scan line, filling in appropriate portions of each line
- Filled portions defined by intersection of scan line and polygon edges
- Runs lying between edges inside P are filled

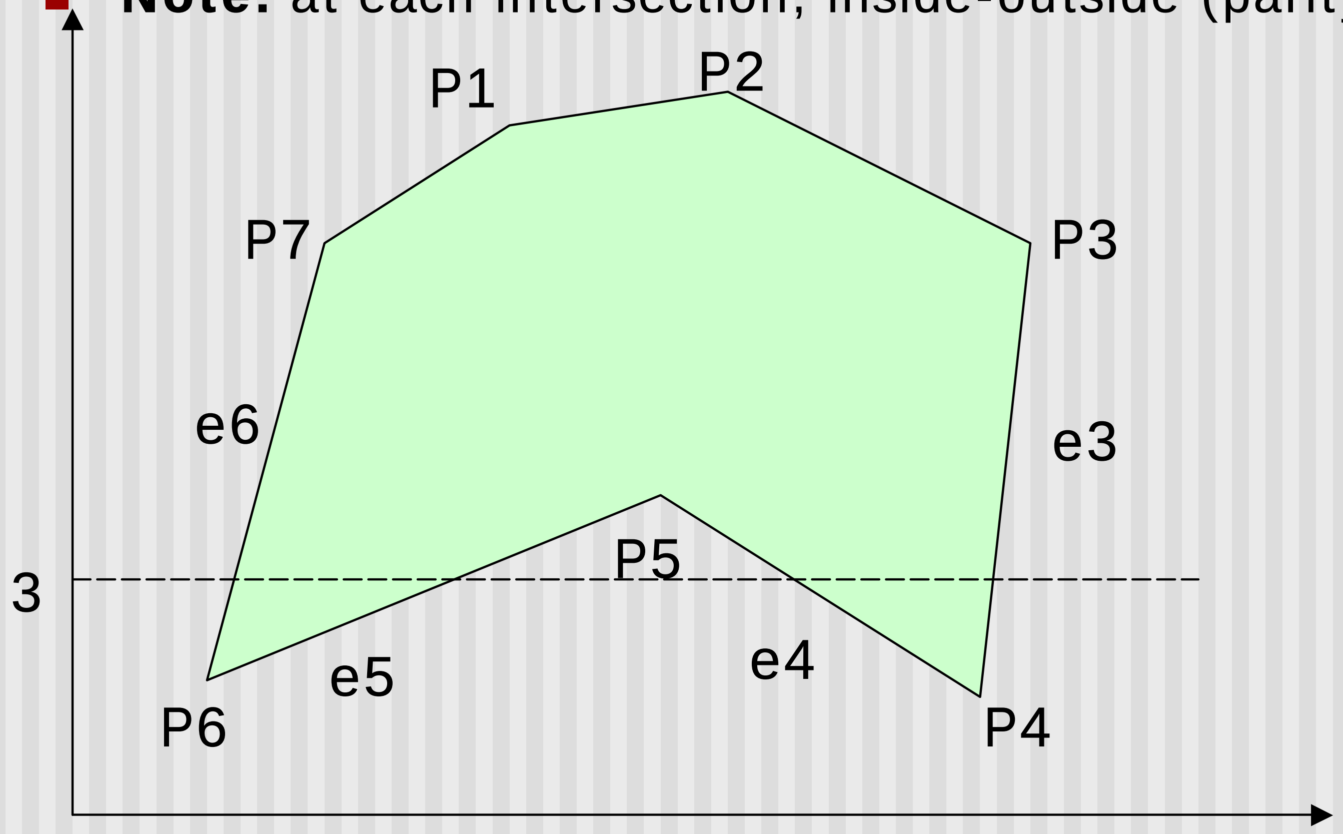
Filling Polygon-Defined Regions

- **Pseudocode:**

```
for(each scan Line L)
{
   Find intersections of L with all edges of P
   Sort the intersections by increasing x-value
   Fill pixel runs between all pairs of
   intersections
}
```

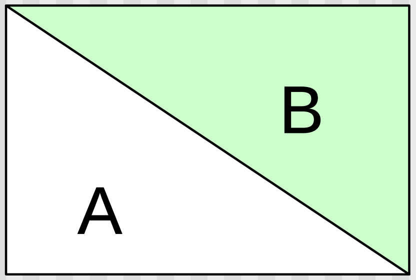
Filling Polygon-Defined Regions

- **Example:** scan line y = 3 intersects 4 edges e3, e4, e5, e6
- Sort x values of intersections and fill runs in pairs
- **Note:** at each intersection, inside-outside (parity), or vice versa



Filling Polygon-Defined Regions

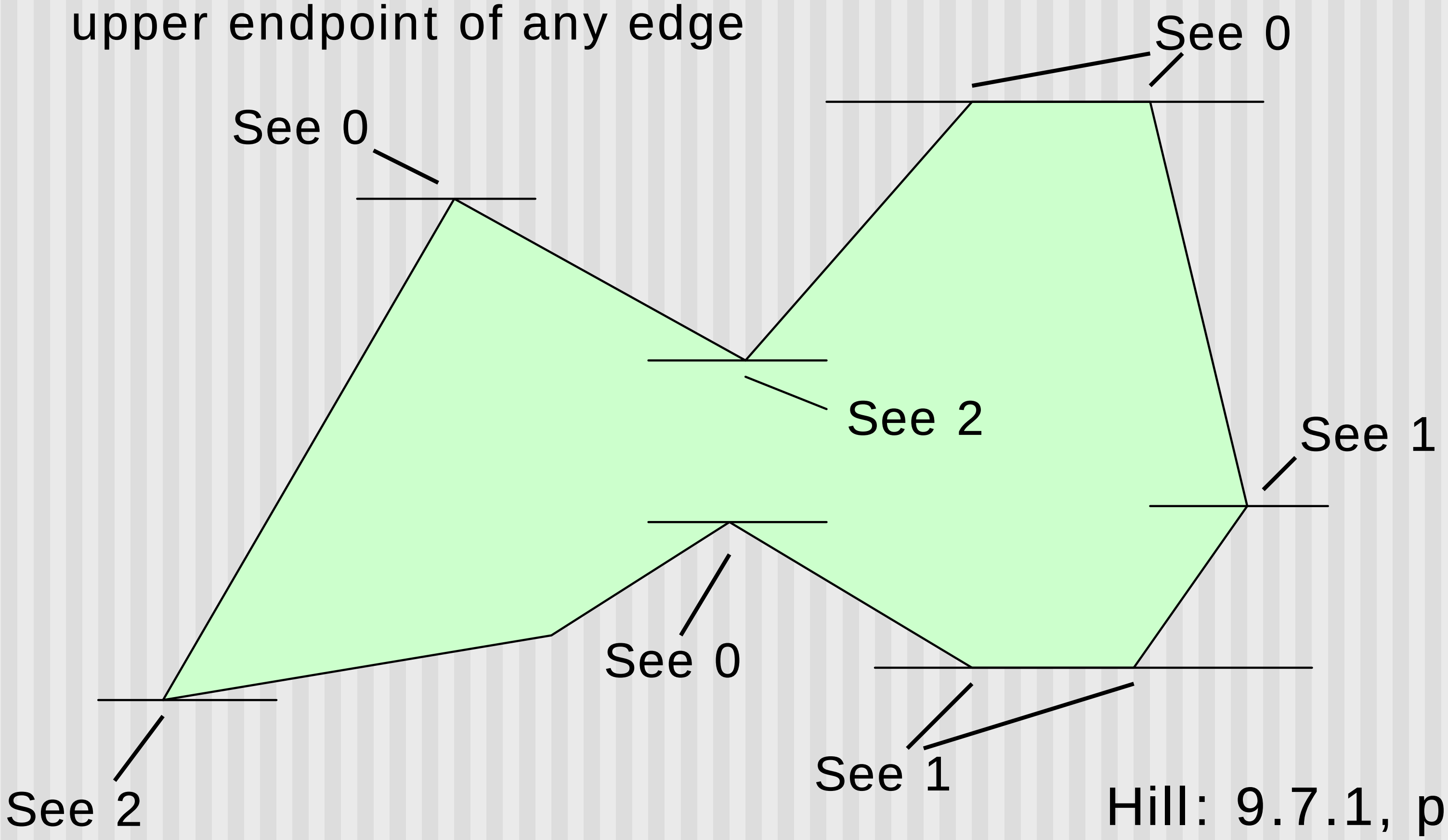
- What if two polygons A, B share an edge?
- Algorithm behavior could result in:
  - setting edge first in one color and the another
  - Drawing edge twice too bright
- **Make Rule:** when two polygons share edge, each polygon owns its left and bottom edges
- E.g. below draw shared edge with color of polygon **B**



Read: Hill: 9.7.1, pg 481

Filling Polygon-Defined Regions

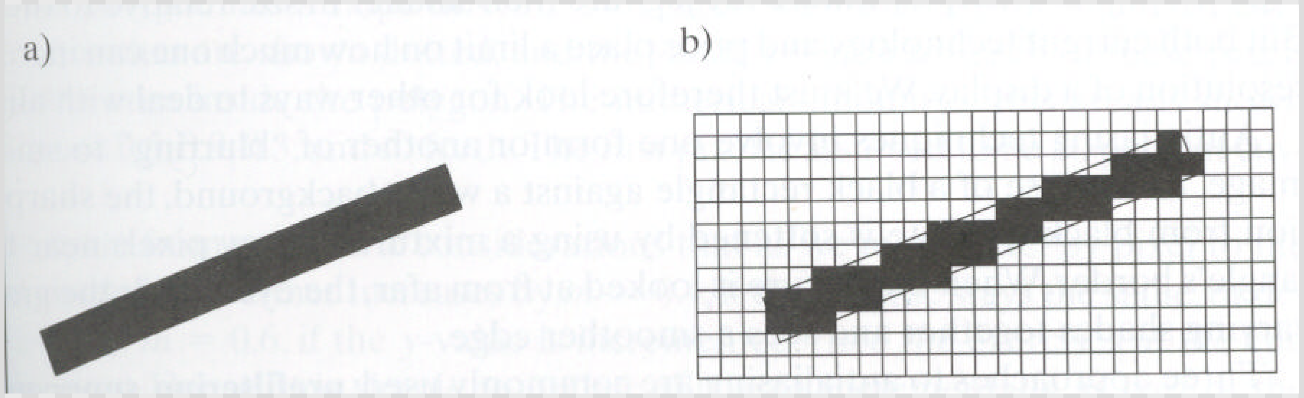
- How to handle cases where scan line intersects with polygon endpoints to avoid wrong parity?
- Solution: Discard intersections with horizontal edges and with upper endpoint of any edge



Hill: 9.7.1, pg. 482

Antialiasing

- Raster displays have pixels as rectangles
- Aliasing: Discrete nature of pixels introduces “jaggies”



Antialiasing

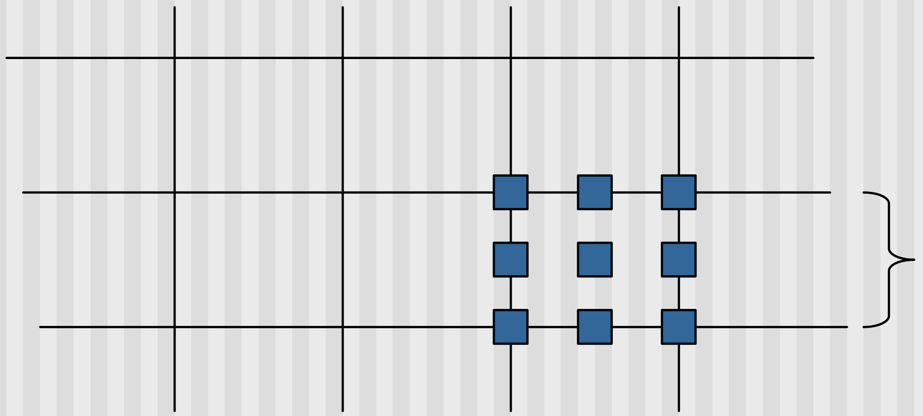
- Aliasing effects:
  - Distant objects may disappear entirely
  - Objects can blink on and off in animations
- Antialiasing techniques involve some form of blurring to reduce contrast, smoothen image
- Three antialiasing techniques:
  - Prefiltering
  - Postfiltering
  - Supersampling

Prefiltering

- Basic idea:
  - compute area of polygon coverage
  - use proportional intensity value
- Example: if polygon covers ¼ of the pixel
  - use ¼ polygon color
  - add it to ¾ of adjacent region color
- Cons: computing pixel coverage can be time consuming

Supersampling

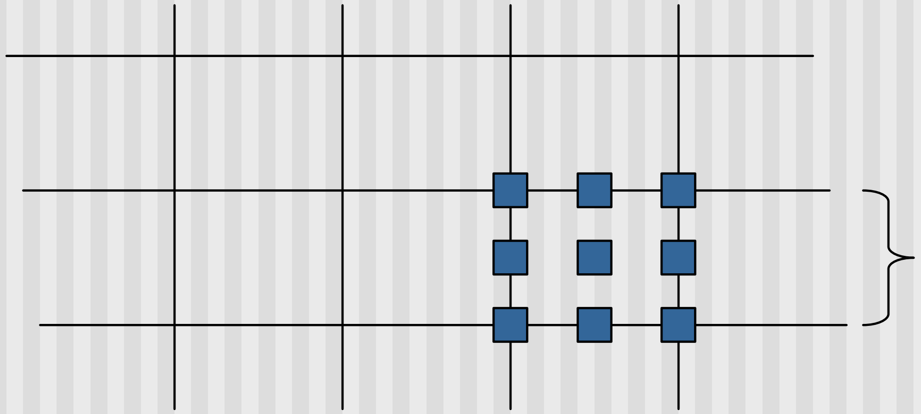
- Useful if we can compute color of any (x,y) value on the screen
- Increase frequency of sampling
- Instead of (x,y) samples in increments of 1
- Sample (x,y) in fractional (e.g. ½) increments
- Find average of samples
- Example: Double sampling = increments of ½ = 9 color values averaged for each pixel



Average 9 (x, y) values to find pixel color

Postfiltering

- Supersampling uses average
- Gives all samples equal importance
- Post-filtering: use weighting (different levels of importance)
- Compute pixel value as weighted average
- Samples close to pixel center given more weight



Sample weighting

1/16	1/16	1/16
1/16	1/2	1/16
1/16	1/16	1/16

Antialiasing in OpenGL

- Many alternatives
- Simplest: accumulation buffer
- Accumulation buffer: extra storage, similar to frame buffer
- Samples are accumulated
- When all slightly perturbed samples are done, copy results to frame buffer and draw

Antialiasing in OpenGL

- First initialize:
  - **`glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH);`**
- Zero out accumulation buffer
  - **`glClear(GLUT_ACCUM_BUFFER_BIT);`**
- Add samples to accumulation buffer using
  - **`glAccum( )`**

Antialiasing in OpenGL

- Sample code
- jitter[] stores randomized slight displacements of camera,
- factor, f controls amount of overall sliding

```
glClear(GL_ACCUM_BUFFER_BIT);
for(int i=0;i < 8; i++)
{
  cam.slide(f*jitter[i], f*jitter[i].y, 0);
  display( );
  glAccum(GL_ACCUM, 1/8.0);
}
glAccum(GL_RETURN, 1.0);
```

jitter.h

-0.3348, 0.4353

0.2864, -0.3934

……

References

- Hill, chapter 9

Interactive Computer Graphics

Lecture 3

Aliasing and Anti-Aliasing
- Hearn & Baker, chapter 4-17

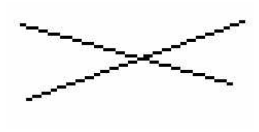
2D transforms
- Hearn & Baker, chapter 5

1

Aliasing and Anti-Aliasing

Lecture 3

Problem: “jaggies”

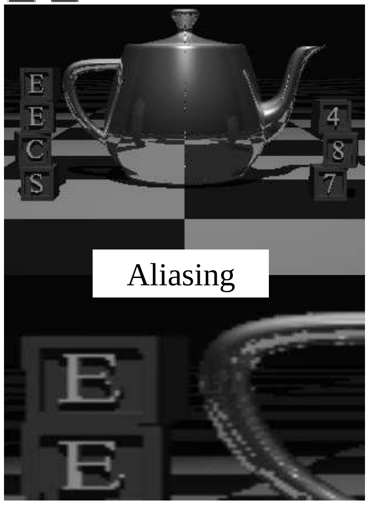


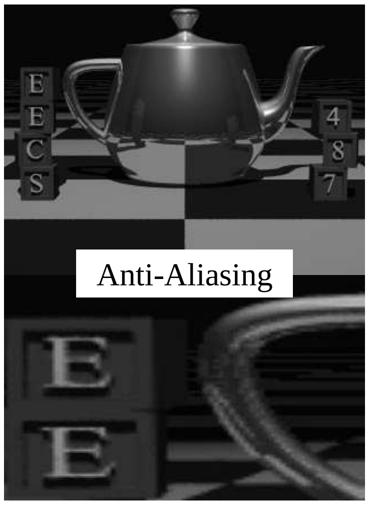
Also known as “aliasing.” It results from sampling a pattern (signal) at limited resolution.

2

Aliasing and Anti-Aliasing -Examples

Lecture 3



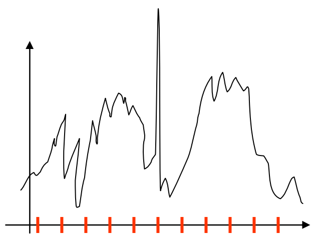


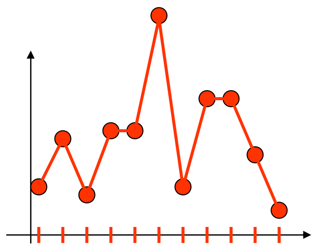
3

Aliasing in photos

Lecture 3

- A piece of practical knowledge
  - Example web page
  - Often better to blur before resizing





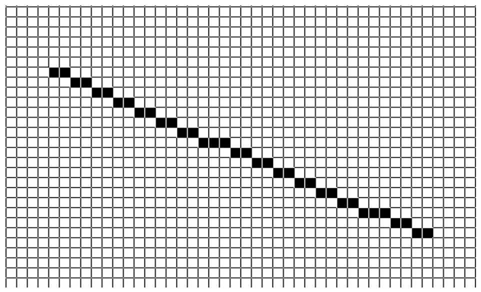
4

Aliasing and Anti-Aliasing

Lecture 3

To fully understand the causes of aliasing requires the use of Signal Processing and Fourier Transforms.

We can look at Aliasing from a more intuitive basis.

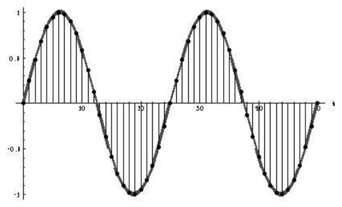


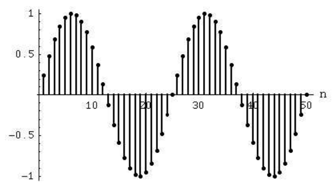
5

Aliasing and Anti-Aliasing

Lecture 3

Effects of Sampling



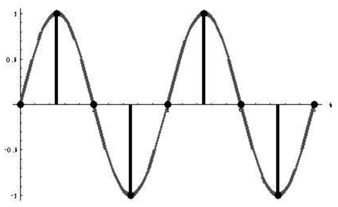


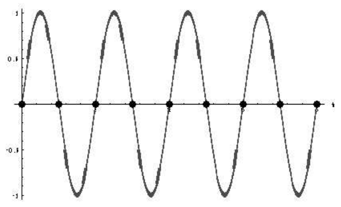
6

Aliasing and Anti-Aliasing

Lecture 3

Effects of Sampling



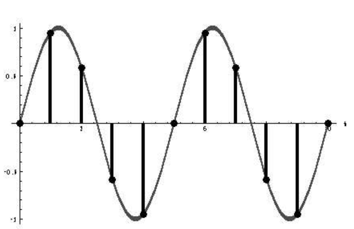


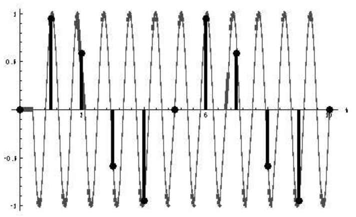
7

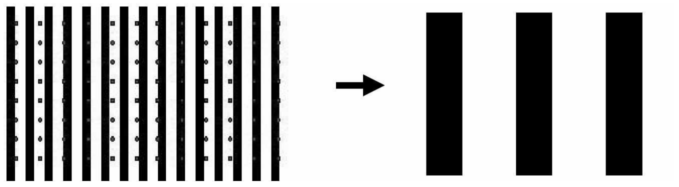
Aliasing and Anti-Aliasing

Lecture 3

Aliasing







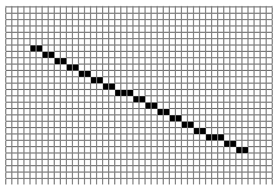
8

Aliasing and Anti-Aliasing

Lecture 3

Anti-Aliasing is the process that attempts to prevent or fix the problem.

- More Resolution
- Unweighted Area Sampling
- Weighted Area Sampling
- Post image creation filtering

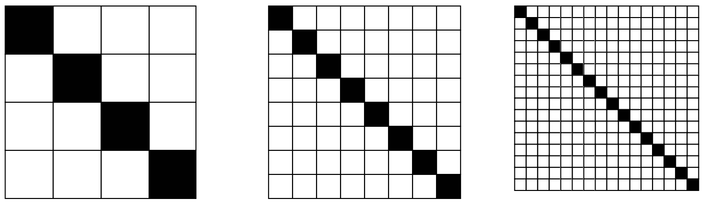


9

Aliasing and Anti-Aliasing

Lecture 3

More Resolution

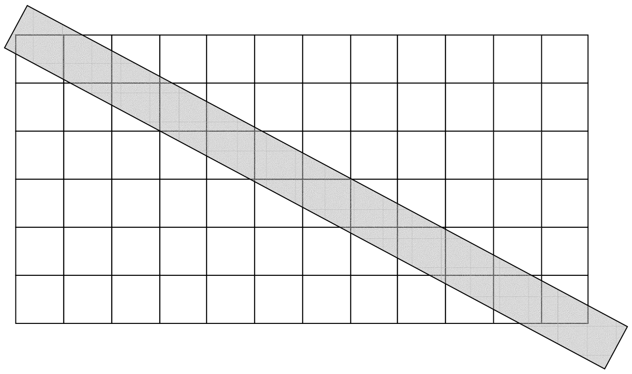


10

Aliasing and Anti-Aliasing

Lecture 3

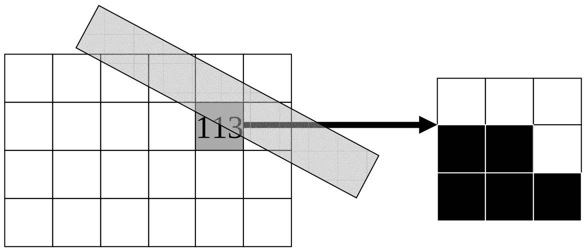
Unweighted Area Sampling



Aliasing and Anti-Aliasing

Lecture 3

Unweighted Area Sampling



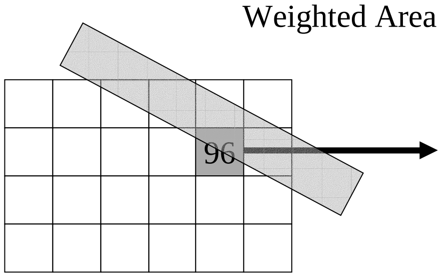
Pixel

Sub Pixels

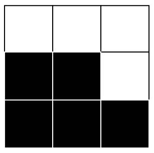
255 * 4/9 = 113

Aliasing and Anti-Aliasing

Weighted Area Sampling



Pixel



Sub Pixels

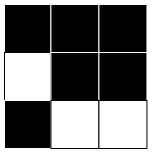
1	2	1
2	4	2
1	2	1

Sub Pixels Weights

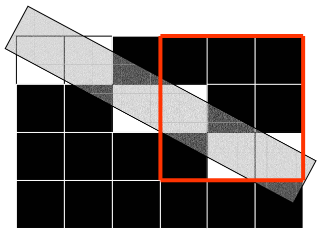
255 * (1+2+1+2)/16 = 96

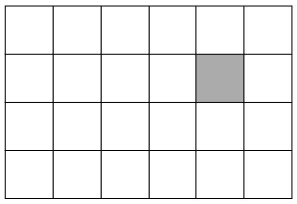
Aliasing and Anti-Aliasing

Post image creation filtering



= 3/9 * 255 = 85





Aliasing and Anti-Aliasing

Lecture 3

Post-image processing:
- fewer samples (less work)
- more blur (worse quality)

Weighted area sampling:
- slightly less blurry than unweighted

15

Matrix Transformations

Lecture 3

Basic Graphics Transforms
- Translation
- Scaling
- Rotation
- Reflection
- Shear

All Can be Expressed As Linear
Functions of the Original Coordinates :

$$x' = Ax + By + C$$
$$y' = Dx + Ey + F$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

16

Affine Transformations

Lecture 3

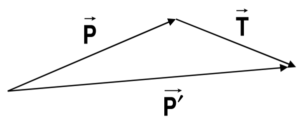
- Preserve Parallel Lines
- Finite Points Map to Finite Points
- Translation, Rotation, and Reflection Preserve :
  - Angles
  - Lengths

17

2-D Transformations - Translation

Lecture 3

$$x' = x + T_x$$
$$y' = y + T_y$$



Vectors : $\vec{P'} = \vec{P} + \vec{T}$

Matrices : $[P'] = [P] + [T]$

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \end{bmatrix} = \begin{bmatrix} P_x + T_x \\ P_y + T_y \end{bmatrix}$$

Transform Polygons by Transforming Each Vertex

18

2-D Transformations - Scaling

Lecture 3

$$x' = S_x \cdot x \qquad [\mathbf{P}'] = [\mathbf{S}][\mathbf{P}]$$
$$y' = S_y \cdot y$$

$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} S_x & 0 \\ 0 & S_y \end{bmatrix} \begin{bmatrix} P_x \\ P_y \end{bmatrix} = \begin{bmatrix} S_x \cdot P_x \\ S_y \cdot P_y \end{bmatrix}$$

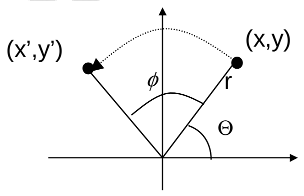
Uniform Scaling : $S_x = S_y$

Differential Scaling : $S_x \neq S_y$

19

2-D Transformations - Rotation

Lecture 3



Rotation about the Origin

$$r\cos\Theta = x$$
$$r\sin\Theta = y$$

$$x' = r\cos(\Theta+\phi) = r\cos\Theta\cos\phi - r\sin\Theta\sin\phi$$
$$y' = r\sin(\Theta+\phi) = r\cos\Theta\sin\phi + r\sin\Theta\cos\phi$$

$$x' = x\cos\phi - y\sin\phi \qquad [\mathbf{P}'] = [\mathbf{R}(\phi)][\mathbf{P}]$$
$$y' = x\sin\phi + y\cos\phi$$

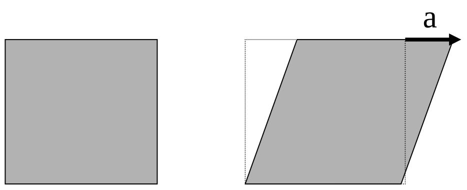
$$\begin{bmatrix} P'_x \\ P'_y \end{bmatrix} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} P_x \\ P_y \end{bmatrix}$$

20

Shear Transforms

Lecture 3

Shear: An action or stress resulting from applied forces that causes two contiguous parts of a body to slide relatively to each other.



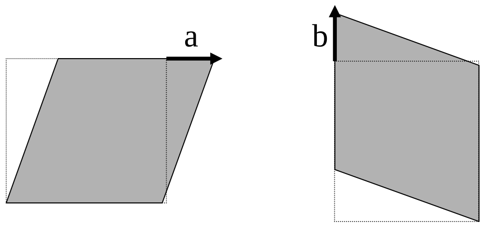
21

Shear Transforms

Lecture 3

$$\begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 \\ b & 1 \end{bmatrix}$$

Shear Translation in X　　Shear Translation in Y



22

Shear Transforms

Properties:

- It is an Affine Transform
- a and b are the proportionality constant

A shear transform application:

- Fast and efficient rotation
- It will be used in perspective transformations

Shear is easily invertable

Shear Transforms

Shear rotation

$$\begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix}\begin{bmatrix} 1 & \lambda \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

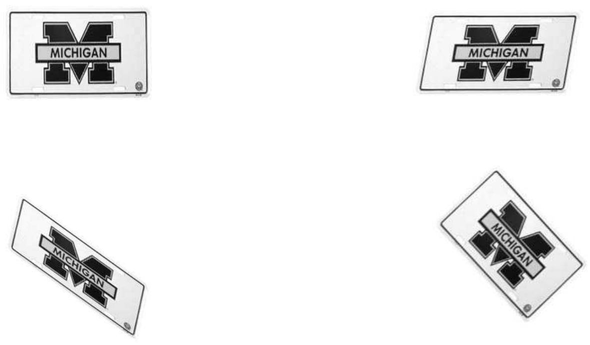
$$\alpha = -\tan\left(\tfrac{\theta}{2}\right)$$

$$\beta = \sin\theta$$

$$\lambda = -\tan\left(\tfrac{\theta}{2}\right)$$

Shear Raster Rotation - Example

Lecture 3



Reflection Transforms

Lecture 3

Reflection is a transform that allows vectors to be flipped about an axis as if reflected in a mirror.

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

Reflection About YZ   Reflection About XZ   Reflection About XY

2-D Transformations

Lecture 3

- Unify Transformation Operations
  - All Transformations Represented as Matrix Products

$$[\mathsf{T}] \equiv \begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \qquad [\mathsf{S}] \equiv \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad [\mathsf{R}(\phi)] \equiv \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$[\mathsf{P}'] = [\mathsf{T}][\mathsf{P}] \qquad [\mathsf{P}'] = [\mathsf{S}][\mathsf{P}] \qquad [\mathsf{P}'] = [\mathsf{R}(\phi)][\mathsf{P}]$$

Homogeneous Coordinates

$$[\mathsf{P}] = \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix}$$

NOT the z coordinate

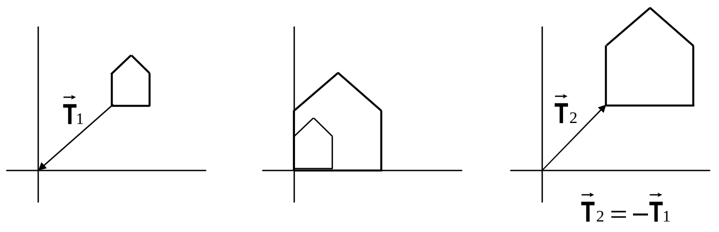
27

Transform Combinations

Lecture 3

- Scaling About a Point
  - Translate to Origin
  - Apply Scale Matrix
  - Translate Back to Original Position

$$[\mathsf{P}'] = [\mathsf{T}_2][\mathsf{S}][\mathsf{T}_1][\mathsf{P}]$$



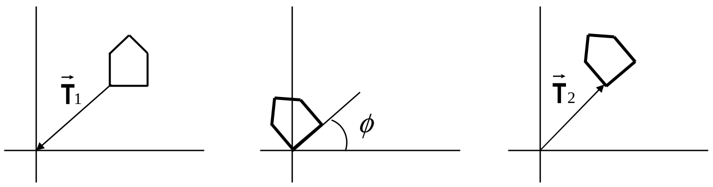
28

Transform Combinations

Lecture 3

- Rotation About a Point
  - Translate to Origin
  - Apply Rotation Matrix
  - Translate Back to Original Position

$$[\mathbf{P}'] = [\mathbf{T}_2][\mathbf{R}(\phi)][\mathbf{T}_1][\mathbf{P}]$$

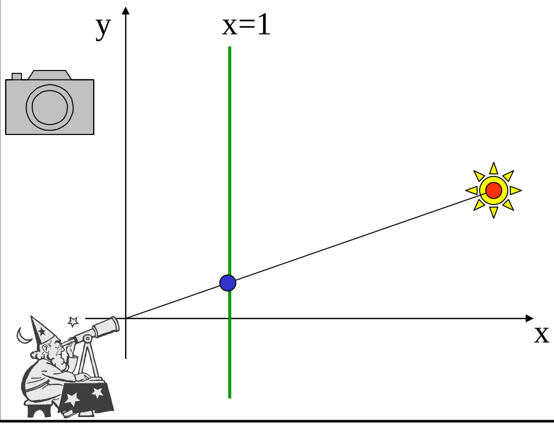


29

Homogeneous coordinates in 2D

Lecture 3

- How to represent perspective projection
  - It is hard!

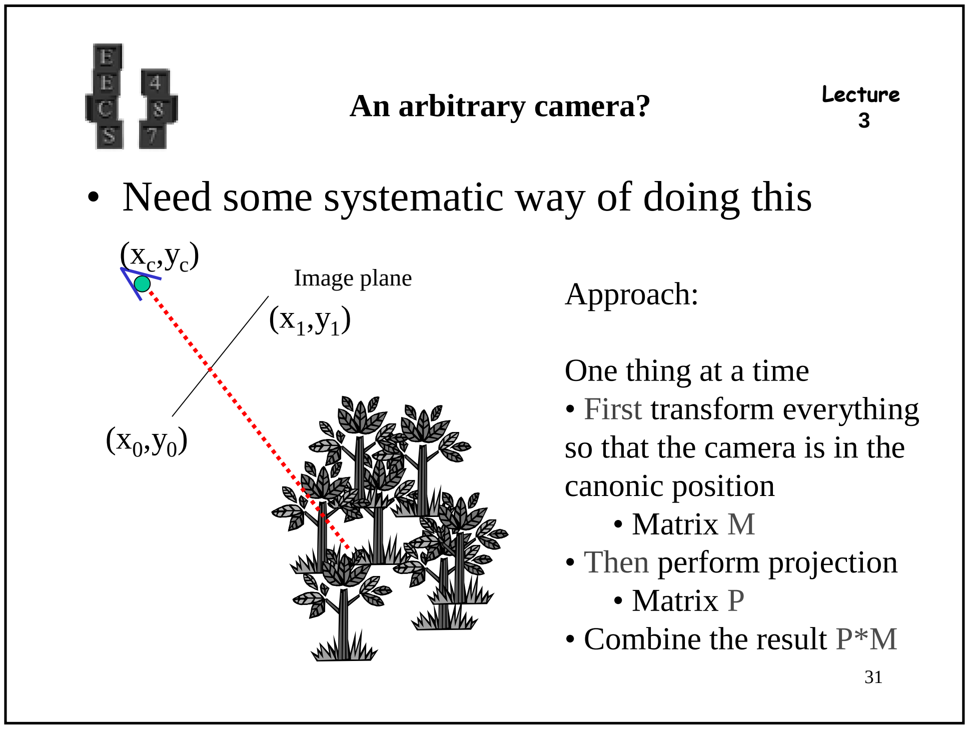


Given the coordinates of the orange point find the coordinates of the blue point

Points: (x,y) $\rightarrow$ (x,y,1)

Equivalency:
(x,y,w) $\equiv$ (x/a, y/a, w/a)

30

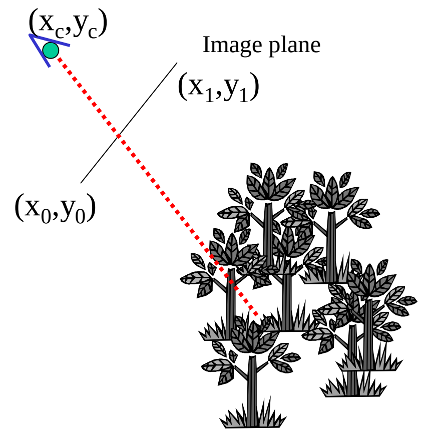


EECS 487

An arbitrary camera?

Lecture 3

- Need some systematic way of doing this



Approach:

One thing at a time

- First transform everything so that the camera is in the canonic position
  - Matrix M
- Then perform projection
  - Matrix P
- Combine the result P*M

31