Lab „Platforms for Embedded Systems“
Chapter 05: 2D Graphics

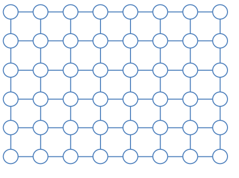
Prof. Dr. Elmar Cochlovius

05: Graphics

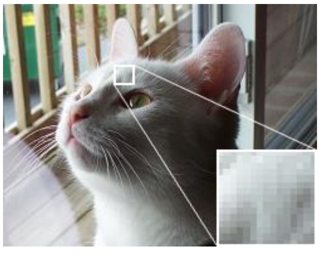
- **Goals of this Chapter:**
  - Understanding Frame Buffers
  - Bresenham‘s Line Algorithm
  - How to accomplish Alphablending
  - Wu‘s Line Algorithm
  - How to do simple Animations

- **Overview**
  - Basics: Overview of Computer Graphics
  - How to look through: Alphablending
  - Putting Pixel on the Display
  - Rasterization: Transforming Shapes into Pixels
  - How to do Antialiasing
  - Simple Animations using Shadow Buffers

- **Basics**
  - What is an IMAGE?
  - Domain **in 2D space** with sampling values at defined and well-ordered points forming a pattern (→„**raster**“)



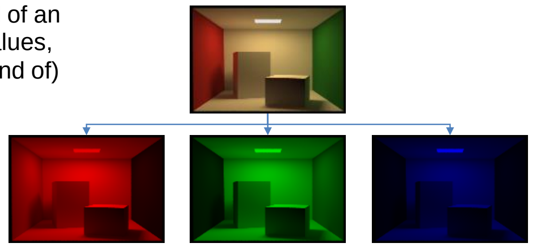
    - Several values per point if necessary
    - Semantics of the values depends on the application (e.g. R,G,B channel, transparency („Alpha“-channel), depth information („Z-Buffer“)
    - Units are int or float → will be translated into voltage values by the display hardware



Basics

- What is a **Channel**?
- **Channel**: all values of an IMAGE which have the same **type**.
- E.g. the Red channel of an Image contains all values, which represent (a kind of) red color.

Original RGB Image
3 Samples per Pixel



Red Channel
1 Sample per Pixel

Green Channell
1 Sample per Pixel

Blue Channel
1 Sample per Pixel

- Well – but there is still something else…

Basics: Alphablending

- A special channel is called **Alpha Channel**
- Next to R,G,B channels, we often use a 4th channel, called α channel to represent transparency / translucency / opacity.
- Values for Alpha: $0 <= \alpha <= 1.0$

The orange box is drawn on top of the purple box using $\alpha = 0.8$

- **Application**: Blending of images
  - High value of α : new pixel has a high degree of opacity
  - Resulting pixel value *C* from old value *A* and new value *B*:

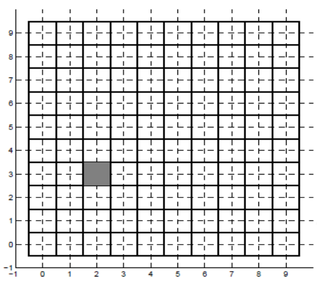
$$C = (\alpha A + (1 - \alpha) B)$$ (linear interpolation)

Overview

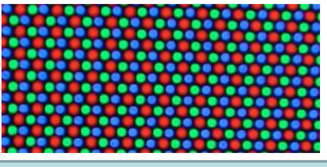
- Basics: Overview of Computer Graphics
- How to look through: Alphablending
- Putting Pixel on the Display
- Rasterization: Transforming Shapes into Pixels
- How to do Antialiasing
- Simple Animations using Shadow Buffers

Basics: Pixel

- Mathematical model of an IMAGE:
  - **Mapping** *n x k* on to a function *u(i,j), 0<= i <n, 0<= j < k*, with interger values *i,j (→ raster)*
- Each pixel value *u(i,j)* is associated with a small „surrounding area“ on the display with the coordinates *(i,j) (→ sampling)*
- Usually, pixel are squared and centered aroung *(i,j).* But this by convention only and depends on the physical hardware:



CRT-TV: rounded Pixels



LCD: rectangular Pixels

Basics: Pixel

- **Frame-Buffer:** area of memory, which values represent the pixel values of the display
  - Number of Pixels: *Width w x Height h*
  - Size [Byte]: *Number of Pixels x Number of Bytes/Pixel*

- **Accesing the fb under Linux**: only via /dev/fb → driver, which manages the frame buffer as a pseudo file.
  - `fbfd= fopen(„/dev/fb0“,…)`

0	1	2	3	4	5	6	7	8	9	...	w-1
w	w+1	w+2	w+3	...							
2w											
3w					3w+5						
4w											
...											
(h-1)w											

- **memory-mapping** is used to „overlay“ the frame buffer with the user-land memory

```
fb-pointer= mmap(…, fbfd,..)
```


Basics: Pixel

- **How are the channels of an IMAGE organized in the frame buffer?**

- **Color models** of the framebuffer device:

- **RGB24** → 8 bits are used for each channel: R,G,B → 3 Bytes/Pixel

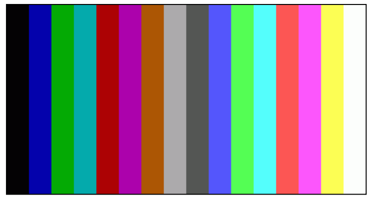
	0			1				w/16			w/16+1			...
0	0	0	0	0	0	0	...	0	0	255	0	0	255	...
1	0	0	0	0	0	0	...	0	0	255	0	0	255	...

- **RGB565** → 5Bit Red + 6Bit Green + 5Bit Blue → 2 Bytes/Pixel

	0		1		2			w/16		w/16+1				...
0	0	0	0	0	0	0	...	0	31	0	31	0	31	...
1	0	0	0	0	0	0	...	0	31	0	31	0	31	...

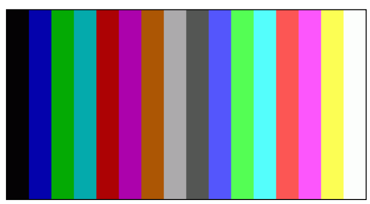
15				11						5					0
r	r	r	r	r	g	g	g	g	g	g	b	b	b	b	b

- **Basics: Pixel**
  - **Color Bars (1/2):**
    To get started, we want to fill the frame buffer based on RGB565 with color bars:
    :



  - **Step 1:**
    open the framebuffer device:
    - `fbfd = open("/dev/fb0", O_RDWR);`
  - **Step 2:**
    Use **io-control** to read the variable and fixed screen parameters (x-Res, y-Res, bits-per-pixel) and save them for later **Restore**
    - `ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo);`
    - `ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo);`
  - **Step 3:**
    calculate the screen size and reserve sufficient mapped memory

```
fbp = (char*)mmap(0, screensize,
        PROT_READ | PROT_WRITE, MAP_SHARED, fbfd, 0);
```

- **Basics: Pixel**
  - **Color Bars (2/2):**
  - **Step 4:** „`draw()` routine“
  - `for all (y-Positions)`

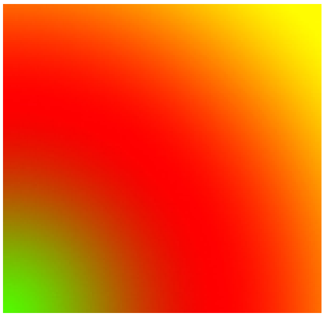
```
for all (y-Positions)
    for all (x-Positions)
          put_pixel_RGB565(x, y, R,G,B);
```

    Note: R,G,B values are predefined in a helper array to get the „right“ colors
  - **Step 5**: „`put-Pixel_RGB565()`-Routine“
    - Calculate byte position in the frame buffer: *pix-offset*
    - Calculate 16Bit *color value*: RRRRR GGG | GGG BBBBB
    - Write *color value* into frame buffer at position *pix-offset*:
    - `unsigned short c = ((r / 8) << 11) +   // Rot-Kanal`

```
unsigned short c = ((r / 8) << 11) +   // Rot-Kanal
                   ((g / 4) << 5) +    // Grün
                    (b / 8);           // Blau
```

    - **Step 6:**
      afterwards, clean up everything to prevent issues w/ virtual console
  - But which color model is used? → `ioctl` bzw. `fbset`

- **Basics: Color Model and Graphics Quality**
  - Question:
    how many different kinds of red tones are available im the RGR565 color model?
  - Question: how can we render a smooth and high-quality color transition, e.g. a nice „sunset“?



- **Basics: Color Model and Graphics Quality**
  - Hopefully NOT like this:

  - But what can we do, if there is only a very limited number of red tones available??

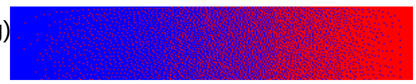
Basics: Graphics Quality and Dithering

- **Dithering** is a common strategy to increase the graphics quality of limited color models. Color transitions are „smudged“.
- Color transition with 300 colors (no Dithering)

- Color transition with 16 colors (after Dithering)

- Color transition with 4 colors (after Dithering)

- Color transition with only 2 colors (after Dithering)



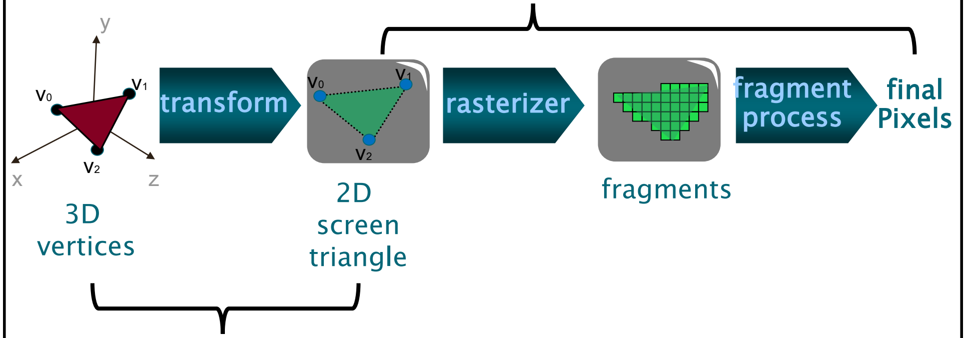
05: Graphics

Basics

- Now we are ready for:
  - **Exercise 12 – Frame buffer, Color Model and Controlling of Pixels**

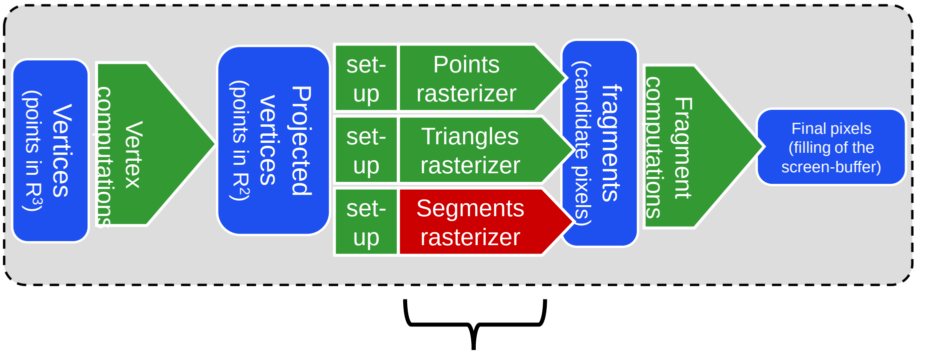
- **Overview**
  - Basics: Overview of Computer Graphics
  - How to look through: Alphablending
  - Putting Pixel on the Display
  - Rasterization: Transforming Shapes into Pixels
  - How to do Antialiasing
  - Simple Animations using Shadow Buffers

- **Rasterization:**
  - **Challenge:** How to transform a **2D scene**, i.e. a collection of 2D primitive shapes, into a „pattern of pixels"?



This is covered in the next chapter
→ 3D topic

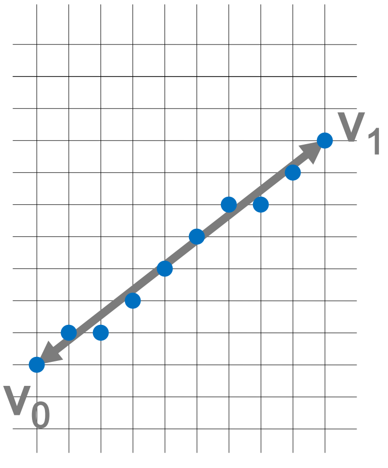
- **Rasterization**
  - **Structure of a simple Graphics Pipeline** (today, often inside a GPU)



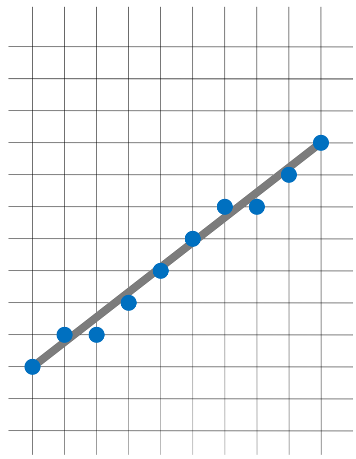
Segments (i.e. short pieces of a line)
are the basic elements!

05: Graphics

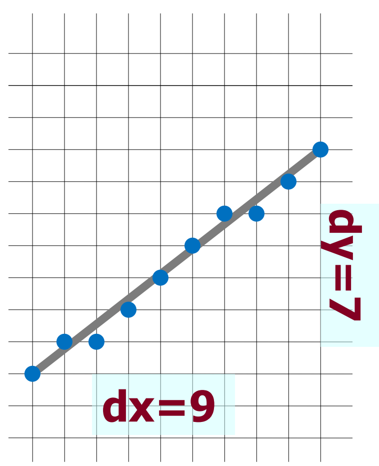
- **Rasterization**
  - **Questions**:
    Which pixels have to be set to
    optimally approximate a line?
  - **Goal**:
    We want to create lines,
    which are as close to the
    „ideal" line as possible.
  - **Constraint**:
    And we want to do it fast!!
    i.e. avoid „expensive" operations,
    e.g. float operations



- **Rasterization**
  - **Task**
    Which sequence of segments (short line fragements) are required to best approximate the original line?
  - **Conclusion:**
    This sequence will have the smallest deviations.



- **Rasterization**
  - **Example:**
    Suppose, slope $|m| \leq 1$
  - We conclude:
    1 fragment per column
  - **Side note:**
    We assume, the line has a line width of 1 pixel.



Rasterization

- The normal form of a line equation is:

  $y = mx + B$

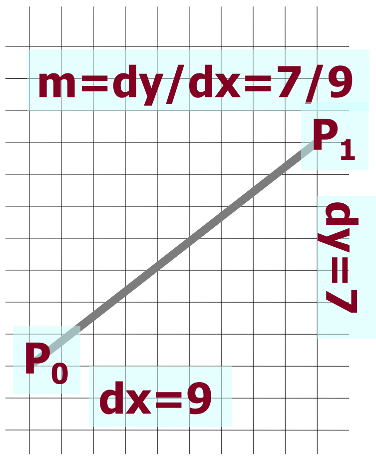
  $(m \leq 1)$

- Rasterization starts at:

  $P_0 = (x_0, y_0)$

  and continues up to

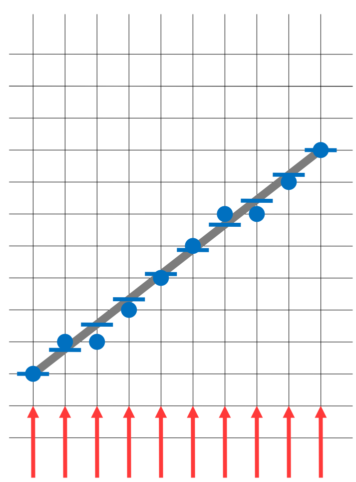
  $P_1 = (x_1, y_1)$



Rasterization

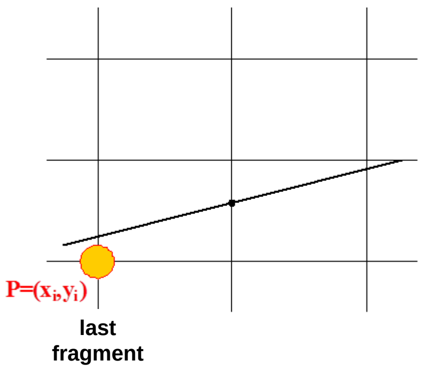
- **„Naive“ analytical approach:**
- Loop over x:
- For x = x0 to x1
  - x ++
  - y ← mx + B
  - round to “next” integer value
  - Output of fragment (x , y)
- **Observation:**
  - 1 addition (integer incr.)
  - 1 multiplikation (float)
  - 1 rounding (float -> int)
- Potential for optimization??

  DDA: Digital Differential Analyzer



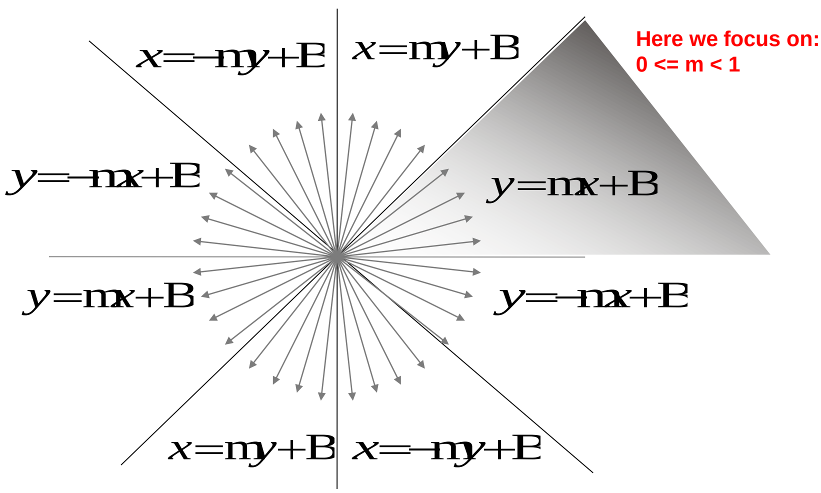
Rasterization: Bresenham‘s line algorithm

- **Goal:**
  Inside the loop, only use integer operations
- **Basic question:**
  Given is the current point P → where to set the next Pixel?
- **Assumption:**
  Again, 0 <= m < 1
- However, this is not a major limitation, since…

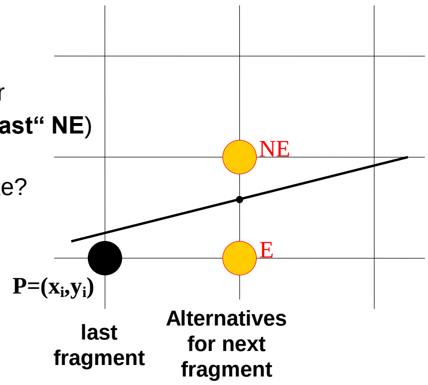


Rasterization: Bresenham‘s line algorithm

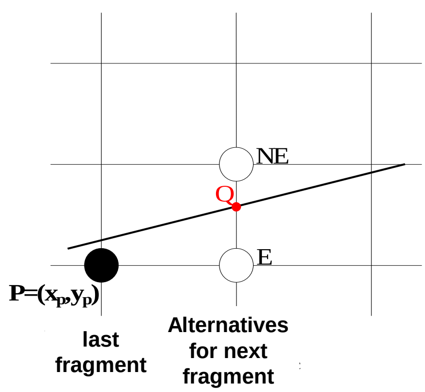
- **Extension to eight Octants:**



- **Rasterization: Bresenham‘s line algorithm**
  - Again:
    Given is the current point P → where to set the next Pixel?
  - Only 2 possible alternatives
    - To the right (**“East“ E**), or
    - To the right-top (**“NorthEast“ NE**)
  - But which one should we take?

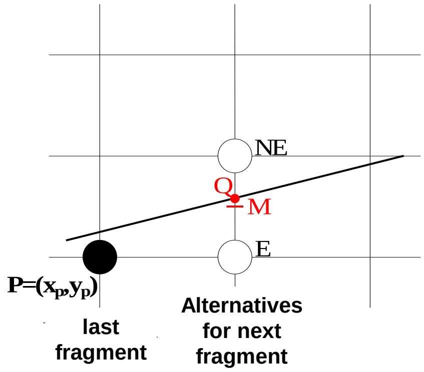


- **Rasterization: Bresenham‘s line algorithm**
  - May **Q** be the **point of intersection** of our line with the next column



Rasterization: Bresenham‘s line algorithm

- May **Q** be the **point of intersection** of our line with the next column
- May **M** be the **center point** of the line E → NE
- This means,
  We need to find out, on which side of M is the intersection point Q, i.e. above or below?



Rasterization: Bresenham‘s line algorithm

- We need to find out, whether Q is above M or below
- Use the **implicit form** of the general line equation

$$F(x, y) = ax + by + c = 0$$

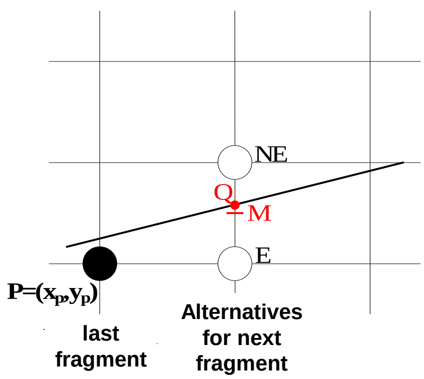
where:

$$m = dy/dx,$$

$$a = dy,$$

$$b = -dx,$$

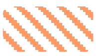
$$c = B \cdot dx$$



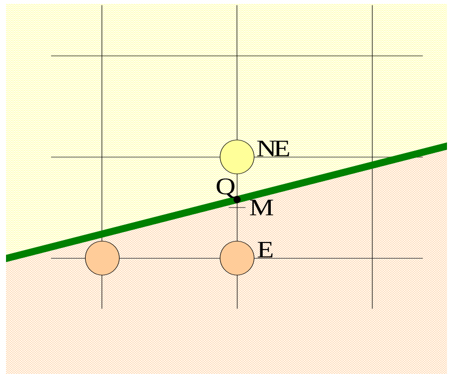
- **Rasterization: Bresenham's line algorithm**
  - Properties of function F:

- **F = 0** for all points ON the line,

- **F < 0** for all points ABOVE the line, and



- **F >0** for all points BELOW the line

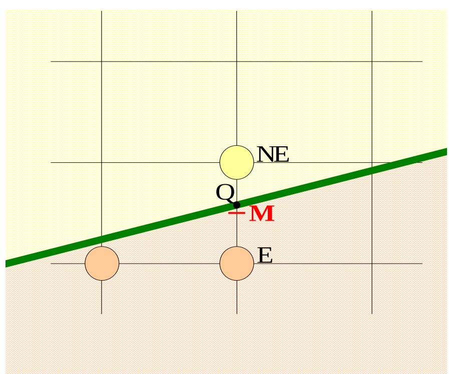


- **Rasterization: Bresenham's line algorithm**
  - This means: to decide between NE and E, it is sufficient to look at the **leading sign** of

$$F(M) = F(x_p + 1, y_p + \frac{1}{2})$$

  - This is done by means of a variable d (**"decision variable"**)

$$d = a(x_p + 1) + b(y_p + \frac{1}{2}) + c$$

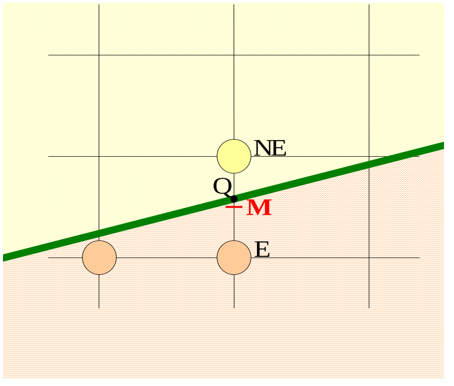


- **Rasterization: Bresenham's line algorithm**
  - This means: to decide between NE and E, it is sufficient to look at the **leading sign** of

$$F(M) = F(x_p + 1, y_p + \frac{1}{2})$$

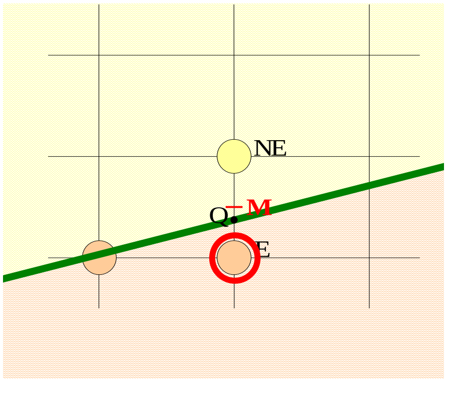
  - This is done by means of a variable d (**"decision variable"**)

$$d = a(x_p + 1) + b(y_p + \frac{1}{2}) + c$$



- **Rasterization: Bresenham's line algorithm**
  - If **d <0** , then **M** is above the line →

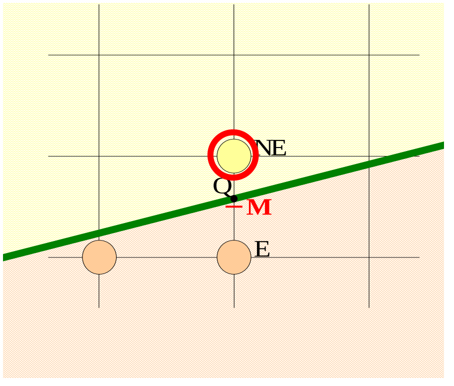
    Point E has to be the next pixel



Rasterization: Bresenham‘s line algorithm

- If **d >0** then
  M is below the line →

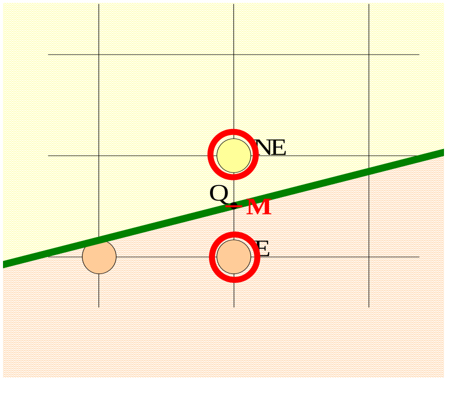
  Point NE is the next pixel



Rasterization: Bresenham‘s line algorithm

- if **d == 0** then
  M is element of the line →

  Point NE OR Point E
  is the next pixel



- **Rasterization: Bresenham‘s line algorithm**
  - Even better news:
    We can calculate d
    incrementally
  - **Example:** we may have choosen point **E**
  - Then we conclude:

$$d_{old} = F(M)$$

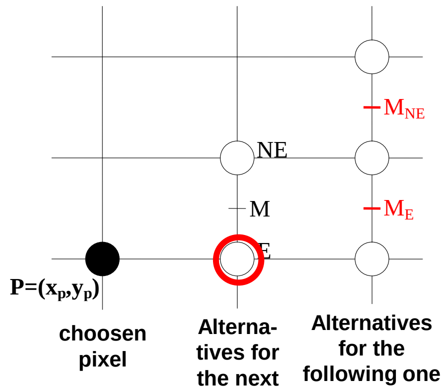
$$d_{new} = F(M_E)$$

  - And therefore:

$$d_{new} = d_{old} + a$$

  - Since we recall:

$$a = dy$$



- **Rasterization: Bresenham‘s line algorithm**
  - Outline:

```
MidpointLine(int x0,
             int y0,
             int x1,
             int y1)
{
int dx, dy, incrE, incrNE, d, x, y;

while (x < x1 ) {
    if ( d <= 0 ) {
        d = d+incrE;
        x++;
    } else {
        d = d+incrNE;
        x++;
        y++;
    }
    OutputFragment(x, y);
}
}
```

Integers

Initialization

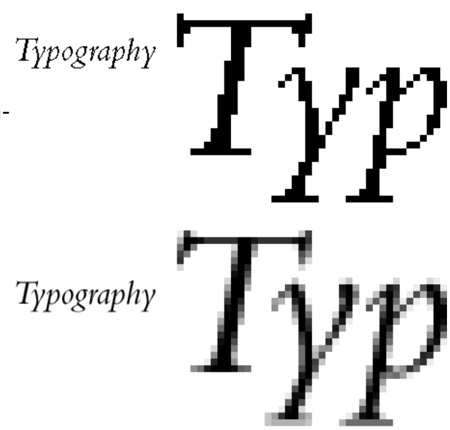
```
dy = y1-y0;
dx = x1-x0;
d = 2*dy-dx;
incrE = 2*dy;
incrNE = 2*(dy-dx);
x = x0;
y = y0;
```

Alternative E

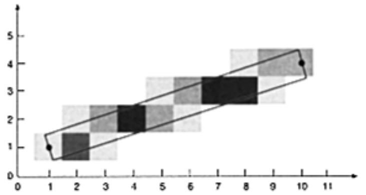
Alternative NE

- **Overview**
  - Basics: Overview of Computer Graphics
  - How to look through: Alphablending
  - Putting Pixel on the Display
  - Rasterization: Transforming Shapes into Pixels
  - How to do Antialiasing
  - Simple Animations using Shadow Buffers

- **Antialiasing**
  - **Problem:** staircase-like artifacts („jaggies") because of unsufficient mapping of con-tinuous, geometric shapes to a discrete 2D pixel raster.
  - **Staircase Effect** can be reduced by **Antialiasing**: Hard edges are softened by calculating average color values:
    - Gray-scale instead of black/white

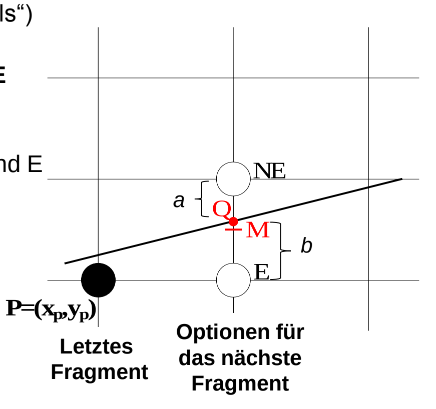


- **Antialiasing**
  - **Implementation approach**:
  - **Area sampling**: the line is approximated by a rectangle, i.e. multiple overlapping pixels
  - **Up to now (Bresenham):** only 1 pixel per column, which is fully set, i.e. 100% black

    Now: pixel intensity is proportional to the size of the overlapping area: 0% (white) <= pixel <= 100% (black**)**
  - 
  - **Note:** usually, >1 pixel per column are affected

05: Graphics

- **Antialiasing: Wu's line algorithm**
  - **Idea**:
  1. Always consider both pixels NE and E („surrounding pixels")
  2. **Q** divides the segment **NE-E** into 2 **parts** *a* and *b*
  3. The pixel intensities of NE and E are inversely proportional to the lengths of *a* and *b*
  4. The sum of NE and E always equals to 100% → why?



- **Graphics: Rasterization and Antialiasing**
  - Now we are ready for:
    - **Exercise 13 –**
      Line Rasterization:
      Bresenham and Wu

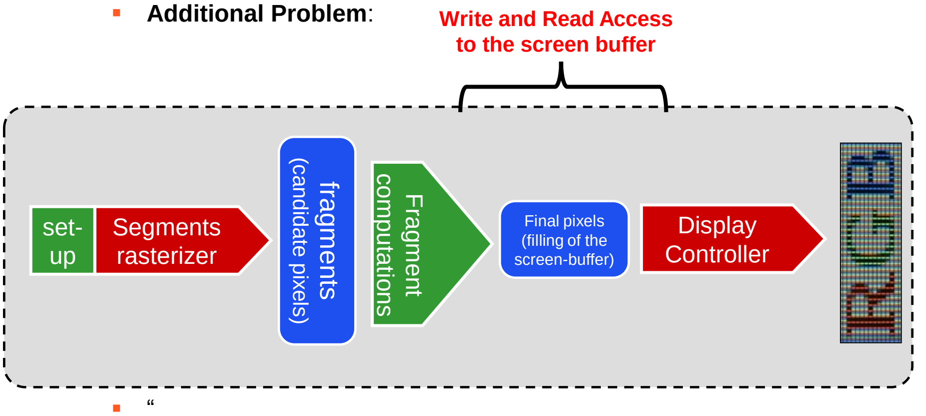
- **Overview**
  - Basics: Overview of Computer Graphics
  - How to look through: Alphablending
  - Putting Pixel on the Display
  - Rasterization: Transforming Shapes into Pixels
  - How to do Antialiasing
  - Simple Animations using Shadow Buffers

- **Animation: Fundamental Problem**
  - **Large amount of data to be calculated and shuffled around in memory:**

  - 1 pixel = up to 32 bit (R,G,B, A) = 4 bytes ("pixel depth")
  - screen buffer = e.g. 1024 x 768 pixels ("screen resolution")
  - frame rate = 60 Hz ("fps")
  - total = 4 x 1024 x 768 x 60 [byte / sec] (**"fill-rate"** in bytes/sec)
  - Results in **188 MBytes / sec**

  - **Conclusion**:
    - Even static graphics content require high data volumes
    - If each frame has a different content, these pixels have to be completely calculated within < 1/60sec.!!
    - Even more challenging: true 3D content **(>>32bit/Pixel**, more complex arithmetics to calculate a „scene")

- **Animation:**
  - **Additional Problem**:



  - "

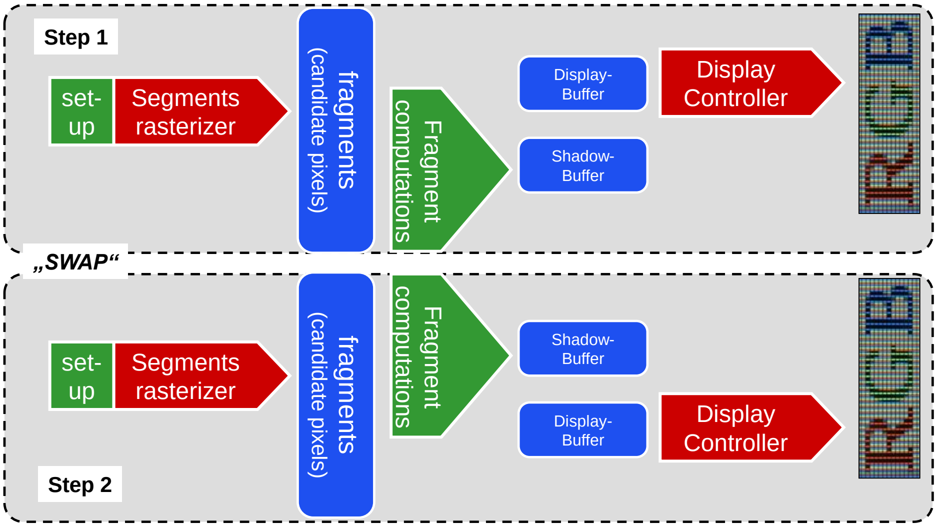
Animation: Screen Tearing

- If several frames have to be displayed in quick sequence („Animation"), then the display contoller may access the screen buffer containing old/incomplete/corrupted content → „screen tearing

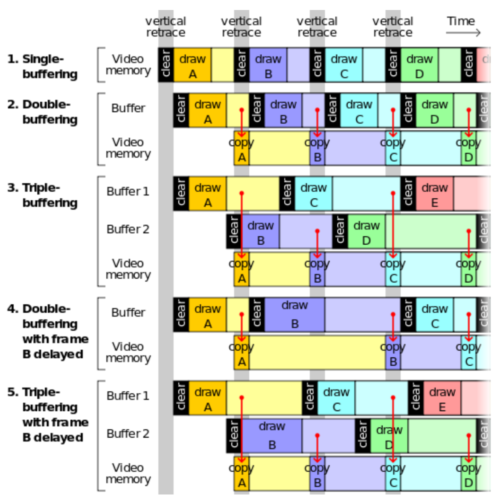
Animation:

- **1st Approach: Use two separate buffers**
- **Step 1:**
  - Buffer A is being displayed (reading from „foreground buffer", „display buffer")
  - Buffer B is used to create the next frame (writing into „background buffer", „shadow buffer")
- After step 1, we switch buffers („**SWAP**")
- **Step 2:**
  - Buffer A is used to create the next frame (writing into „background buffer", „shadow buffer")
  - Buffer B is being displayed (reading from „foreground buffer", „display buffer")

▪ Animation: Double-Buffering



▪ Animation: Triple Buffering and more…

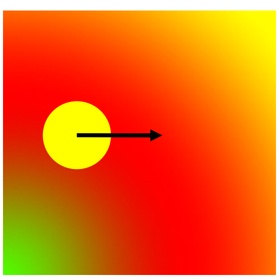


 [Ref: Wikipedia]

- **Animation:**
  - **Constraints for Double Buffering:**
    - Requires Hardware Support
    - Software-only Solution is too „expensive“ / slow
  - **Here, we follow a 2nd approach: „Sprites“**
    - „small“ bitmaps, which are animated on top of a constant background
    - Easier for plattform-independent SW development, since little or no HW dependencies
    - Challenge:
      What do we have to do, to avoid drawing the complete background (static!) for each frame? → Performance issue!

- **Animation:**
  - **Naive Approach: Redraw Sprite on each new position:**

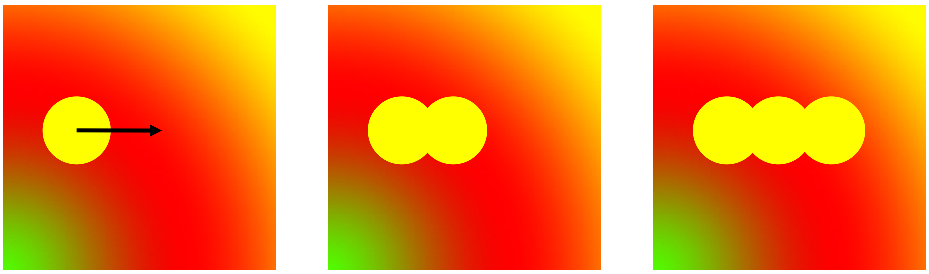
t



Frame 1

05: Graphics

- **Animation:**
  - **Naive Approach: Redraw Sprite on each new position:** *t*



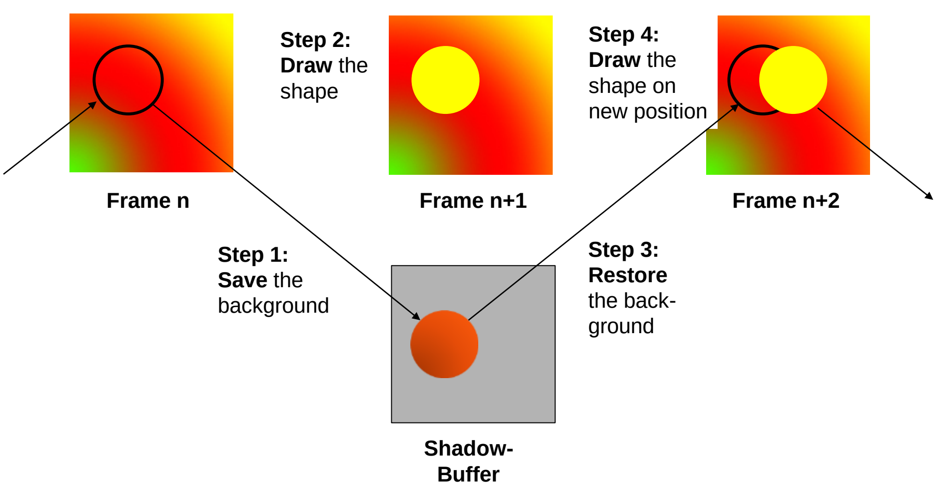
Frame 1:
t0

Frame 2:
t0 + 1/60s.

Frame 3:
t0 + 2/60s

05: Graphics

- **Animation:**
  - **Better approach: save the background before you draw on it!** *t*



Animation:

- **Pseudo Code for simple Animation:**

  For (Start-Position <= Pos <= EndPos) {

  if (Pos > StartPos) then **restoreShape**(an alter Pos.) // all iterations but the very first

  saveShape(an neuer Pos.) // save the background

  drawShape(an neuer Pos.)

  }

- **Comment**:
  - restoreShape() requires restorePixelRGB565(), etc.
  - saveShape() requires savePixelRGB565(), etc.
  - drawShape() requires drawPixelRGB565(), etc.

  - Which of these functions require the effort of Alphablending?

Summary of 2D Graphics:

- Insights into Frame Buffers, Color Models and Pixels
- Bresenham‘s Line Algorithm
- How to accomplish Alphablending and Antialiasing
- Wu‘s Line Algorithm
- How to do simple Animations: Double Buffering vs. Sprites

- **Animation:**
  - Now it is time for:
    - **Exercise 14 –**
      Simple Animations with Sprites

- **Some Hints regarding the Exercises:**
  - During lab hours we will NOT be able to complete all exercises
  - Pls. continue with the exercises as a homework
  - Using the course material and code examples provided, you are in a position to solve these yourself
  - As a minimal solution, pls. solve exercises 12.3-4, 13.1-3, 14.1-4. The rest is regarded as a bonus.

- **References and add. Information:**
  - Applets: http://graphics.cs.brown.edu/research/exploratory/freeSoftware/repository/edu/brown/cs/exploratories/applets/colorMixing/additive_color_mixing_guide.html
  - 2D, Image, http://cs.brown.edu/courses/cs123/lectures.html
  - https://sites.google.com/site/marcoschaerfcomputergraphics/course-notes
  - http://cggmwww.csie.nctu.edu.tw/courses/index.php?course=cgu&year=2008
  - Bresenham https://sites.google.com/site/marcoschaerfcomputergraphics/course-notes/8.1-rendering.ppt?attredirects=0&d=1
  - http://de.wikipedia.org/wiki/Dithering_%28Bildbearbeitung%29http://de.wikipedia.org/wiki/Floyd-Steinberg-Algorithmus
  - Convolution, etc: http://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm