Applications of Computer Graphics

- Computer Art: Using computer graphics we can create fine and commercial art which include animation packages, paint packages. ...
- Computer Aided Drawing: ...
- Presentation Graphics: ...
- Entertainment: ...
- Education: ...
- Training: ...
- Visualization: ...
- Image Processing:

Computer Graphics

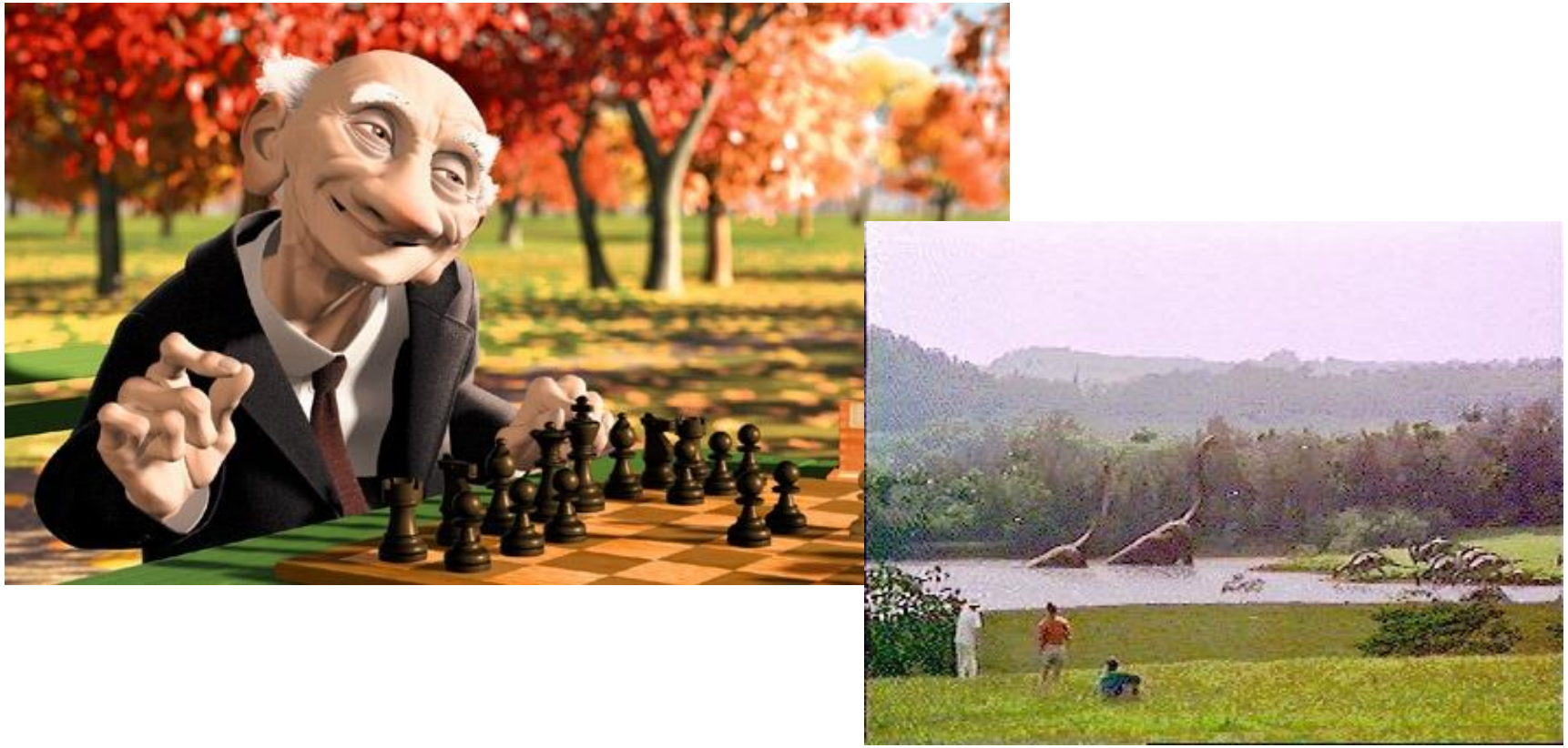
An Introduction

What’s this course all about?

We will cover…

- Graphics programming and algorithms
- Scanconversion algorithms
- Color and Shading
- Applied geometry (Curves and surfaces)
- modelling and rendering

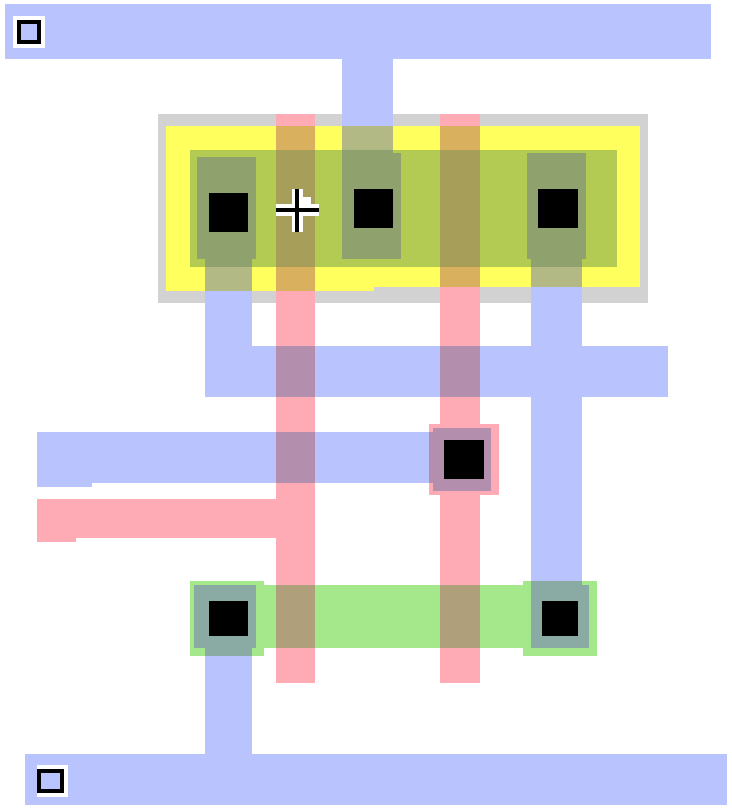
Computer Graphics is about animation (films)

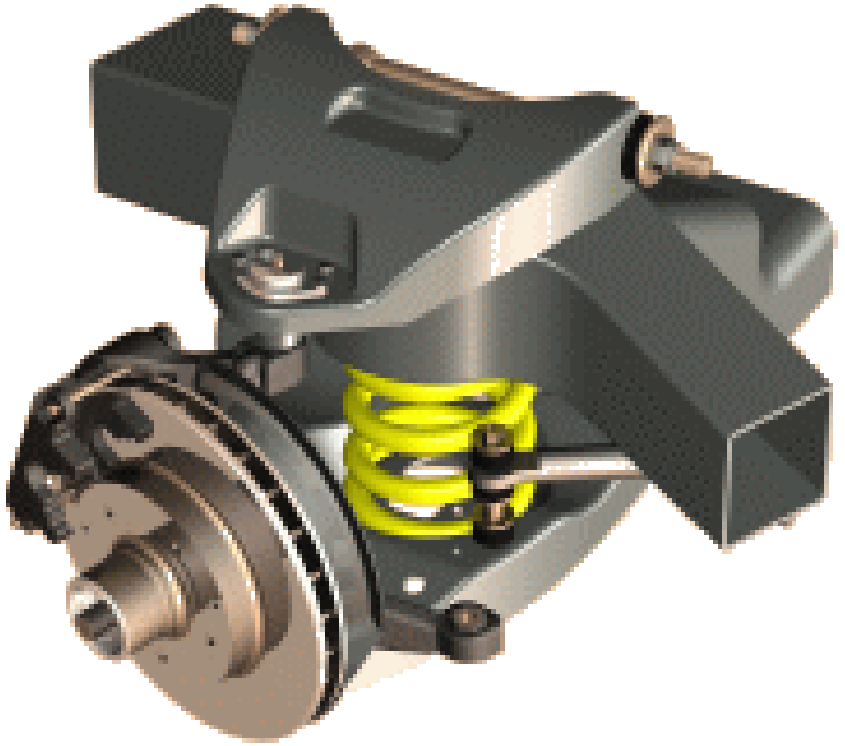


Games are very important in Computer Graphics

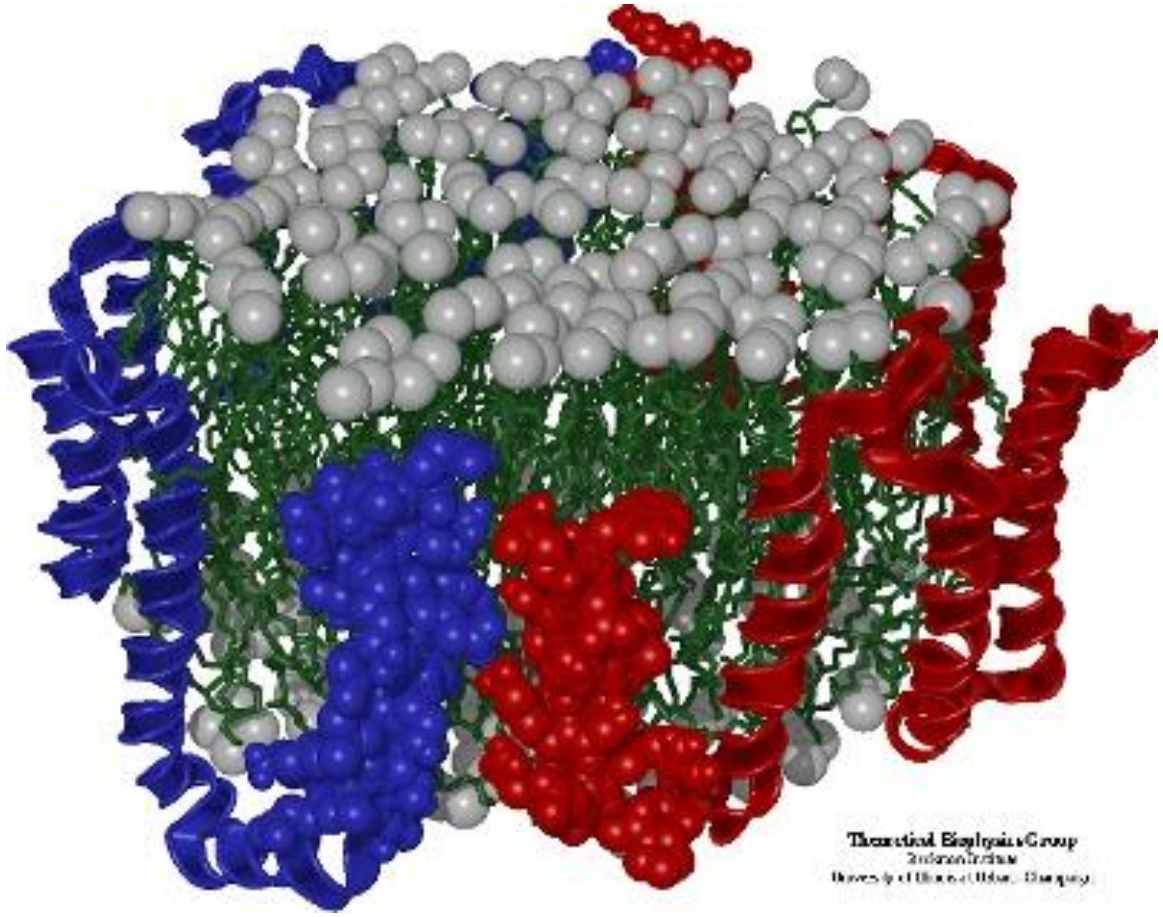
Medical Imaging is another driving force

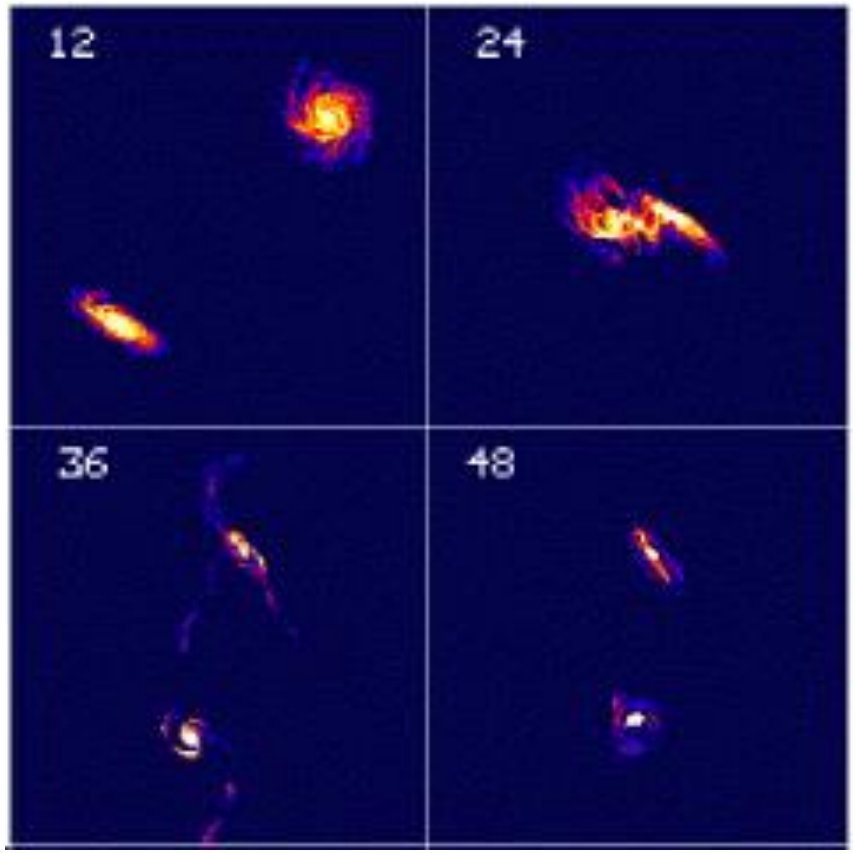
Computer Aided Design too





Scientific Visualisation





Overview of the Course

Graphics Pipeline (Today)

Modelling

    Surface / Curve modelling

(Local lighting effects) Illumination, lighting, shading, mirroring, shadowing

Rasterization (creating the image using the 3D scene)

Ray tracing

Global illumination

Curves and Surfaces

Graphics/Rendering Pipeline

1. Graphics processes generally execute sequentially

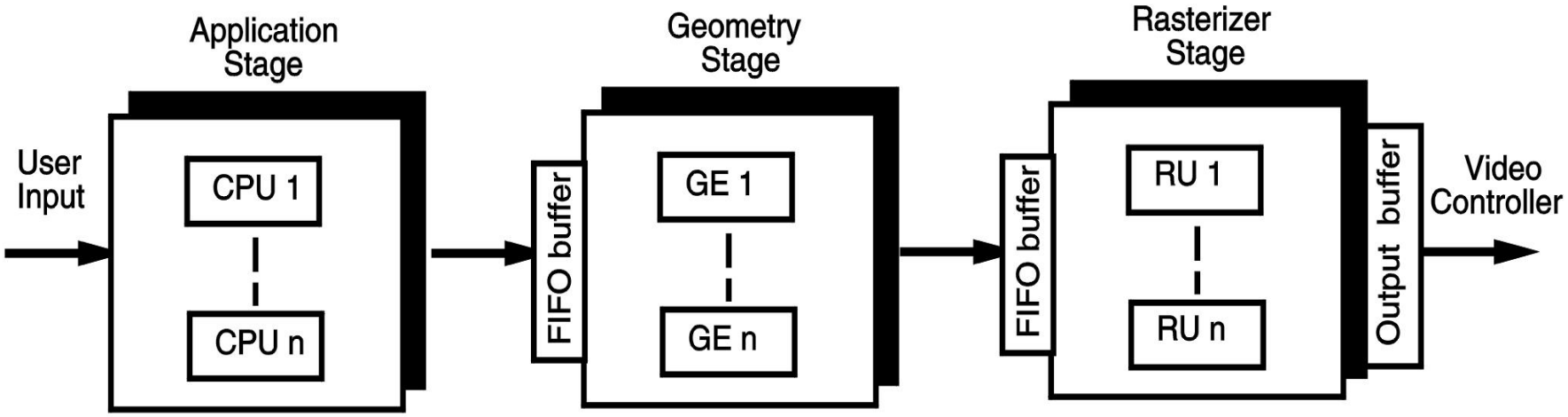
2. Pipelining the process means dividing it into stages

3. Especially when rendering in real-time, different hardware resources are assigned for each stage

Graphics / Rendering Pipeline

There are three stages

1. Application Stage
2. Geometry Stage
3. Rasterization Stage



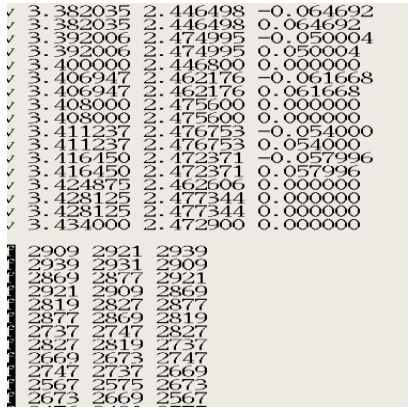
Application stage

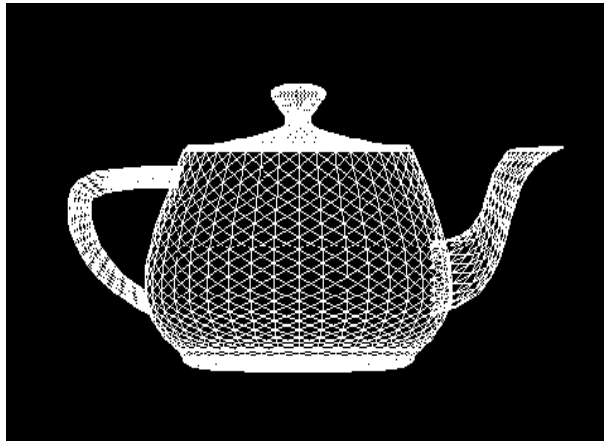
Entirely done in software by the CPU

Read Data

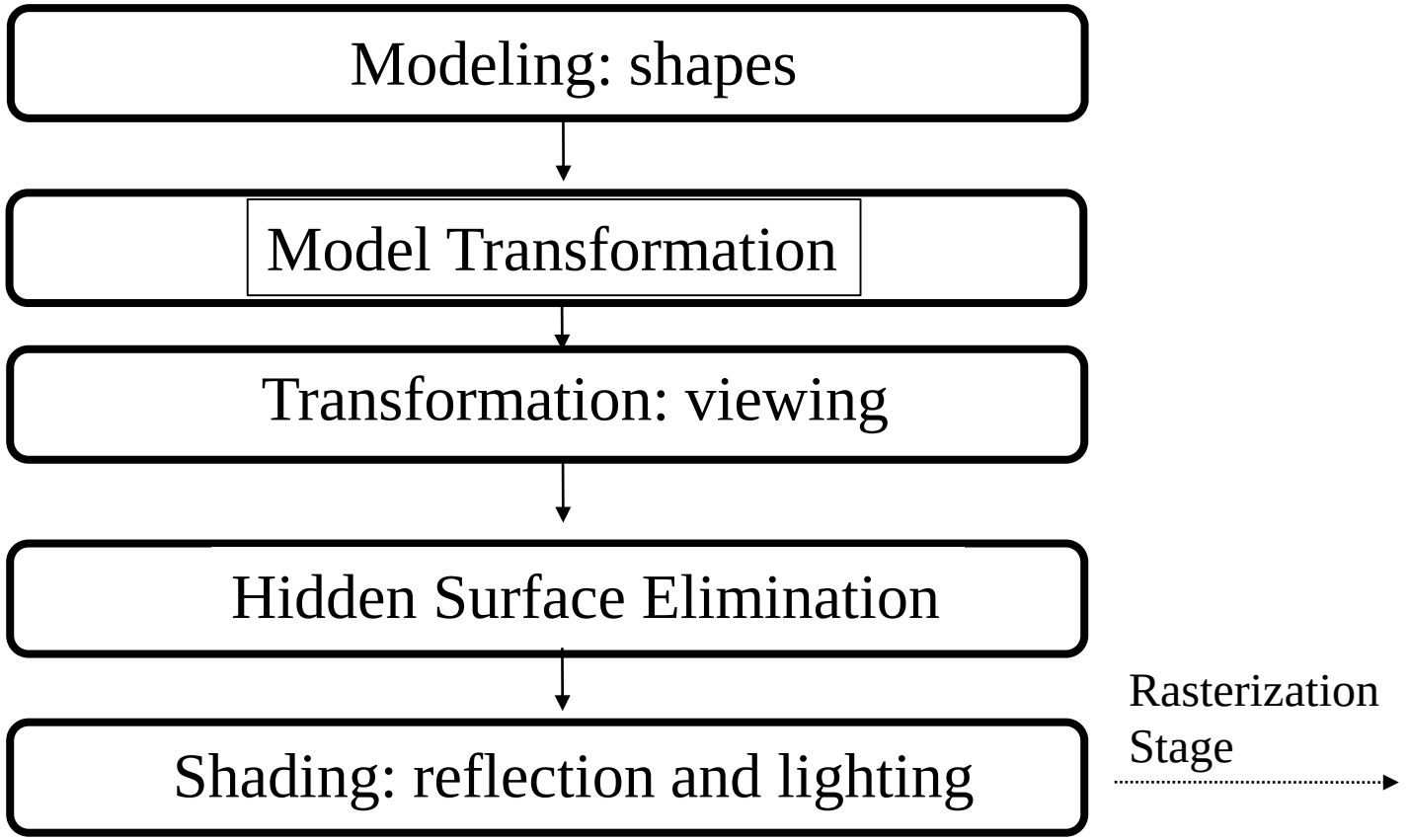
- the world geometry database,
- User's input by mice, trackballs, trackers, or sensing gloves

In response to the user's input, the application stage change the view or scene

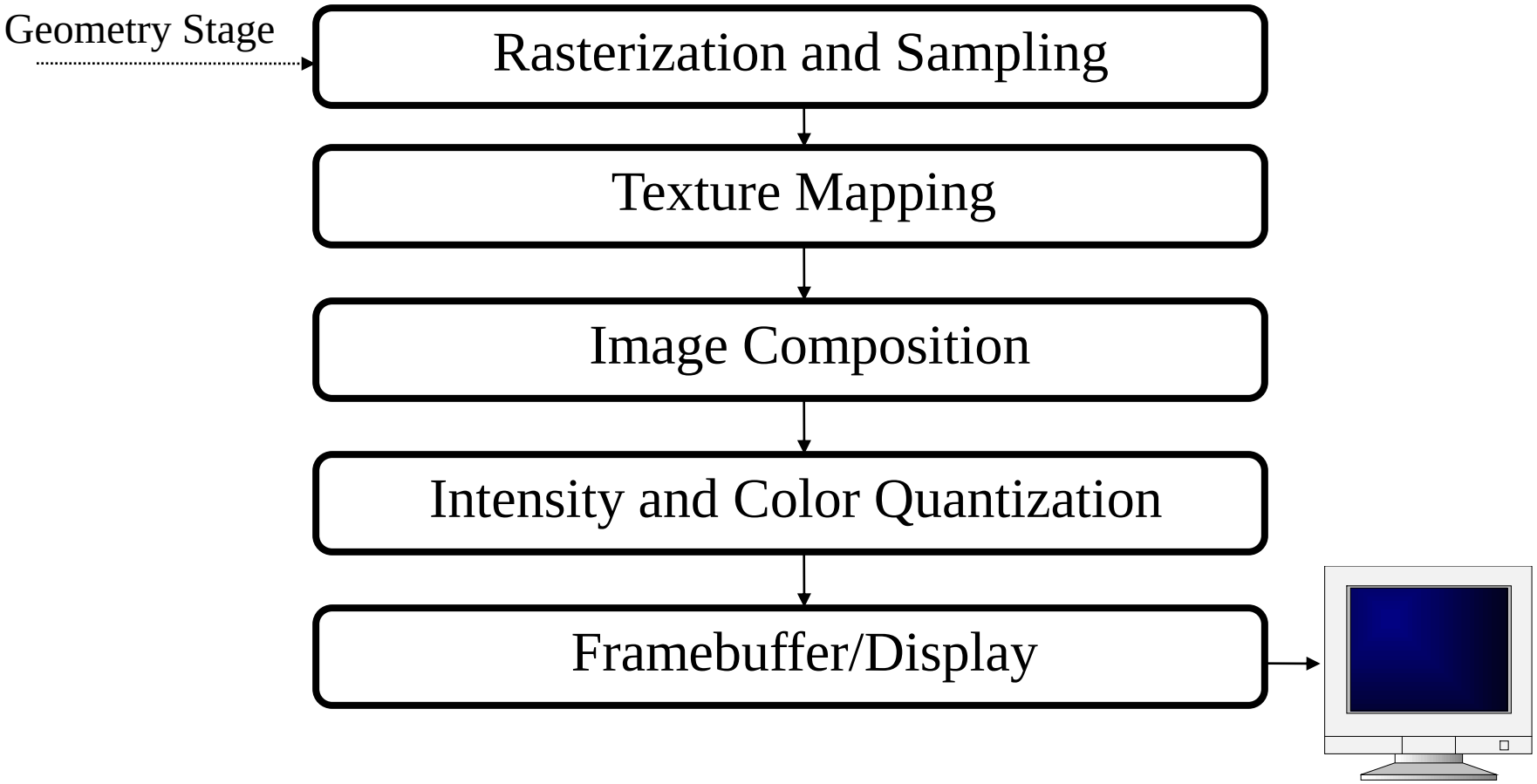




Geometry Stage

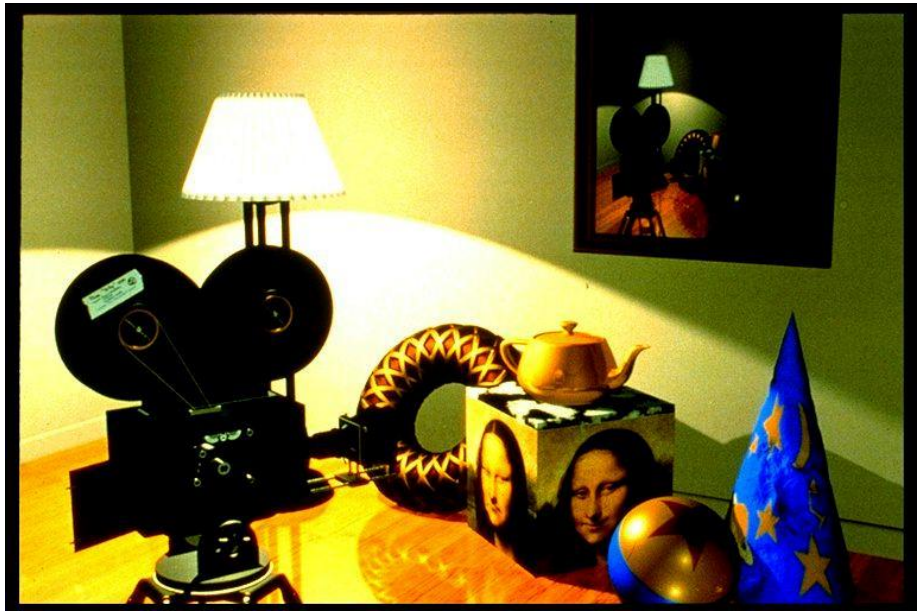


Rasterization Stage

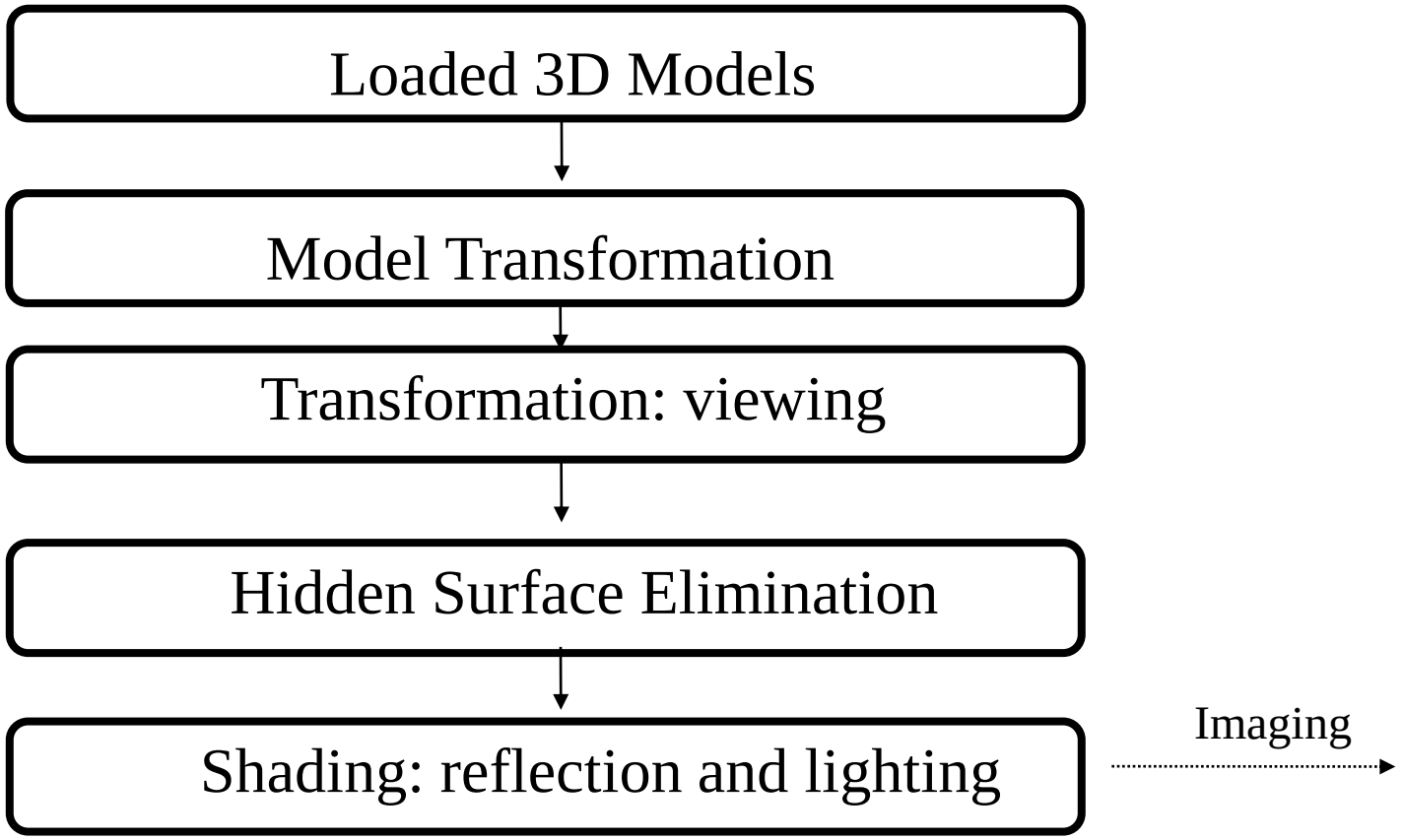


An example thro’ the pipeline…

The scene we are trying to represent:



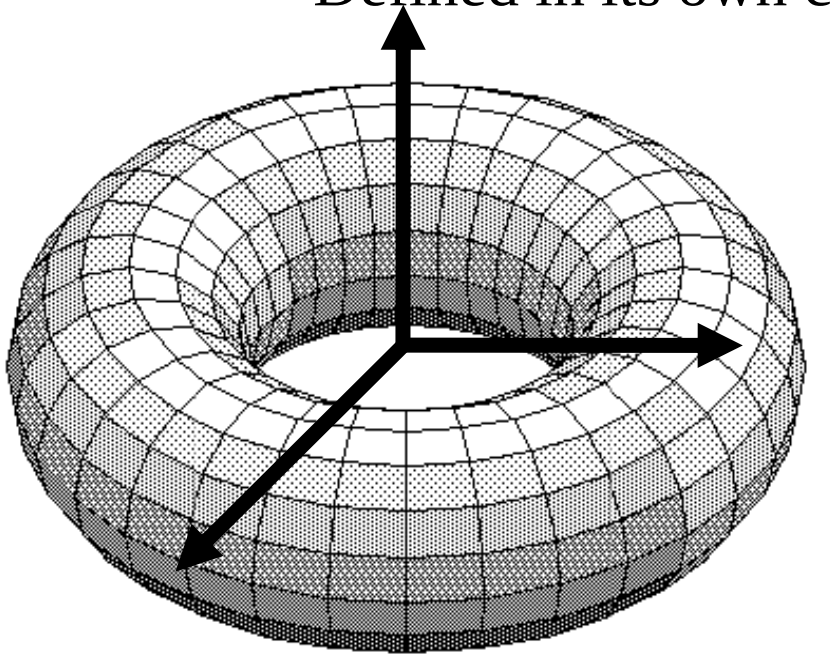
Geometry Pipeline

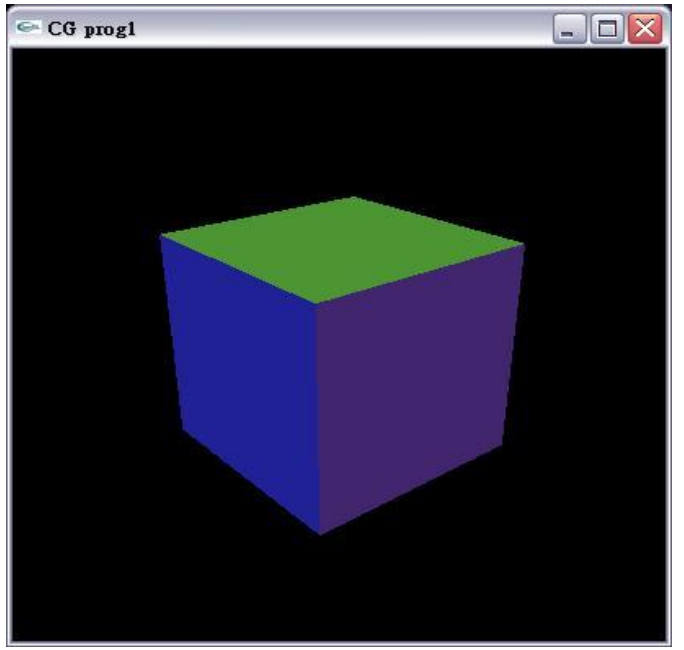


Preparing Shape Models

Designed by polygons, parametric curves/surfaces, implicit surfaces and etc.

Defined in its own coordinate system



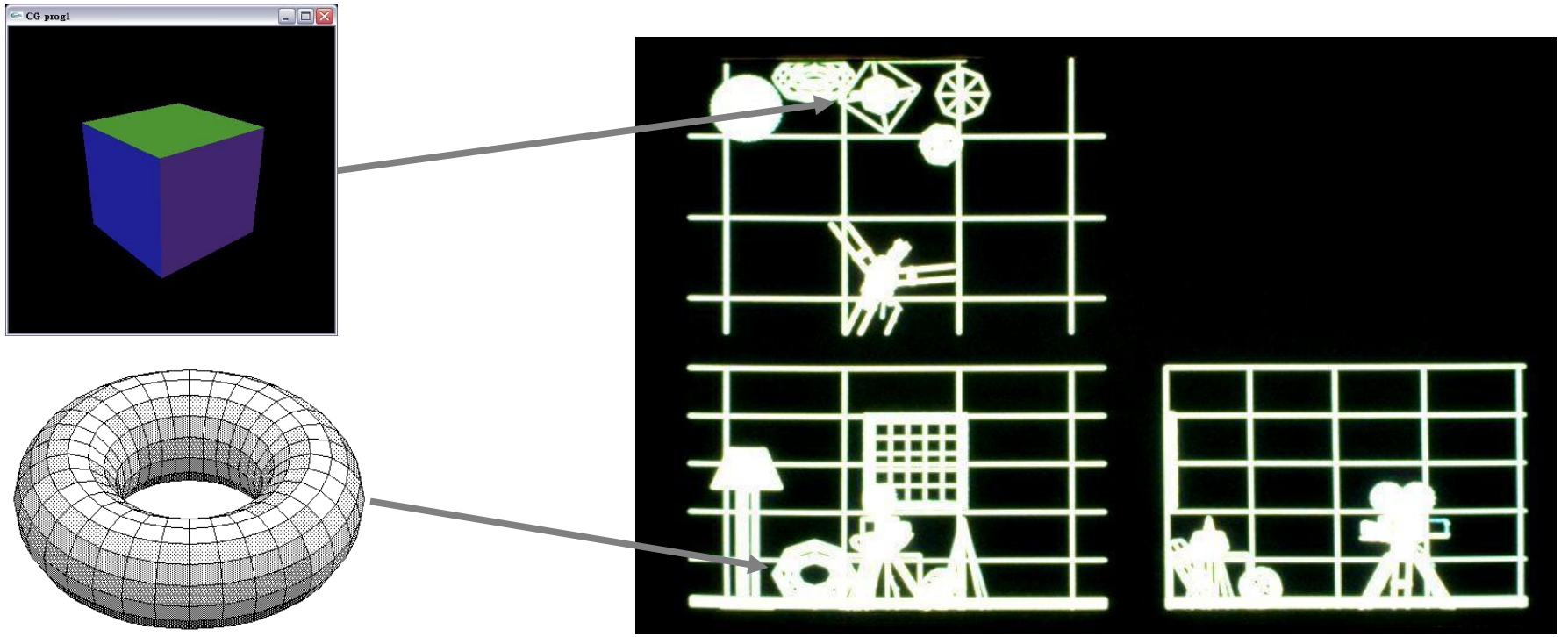


Model Transformation

Objects put into the scene by applying translation, scaling and rotation

Linear transformation called homogeneous transformation is used

The location of all the vertices are updated by this transformation

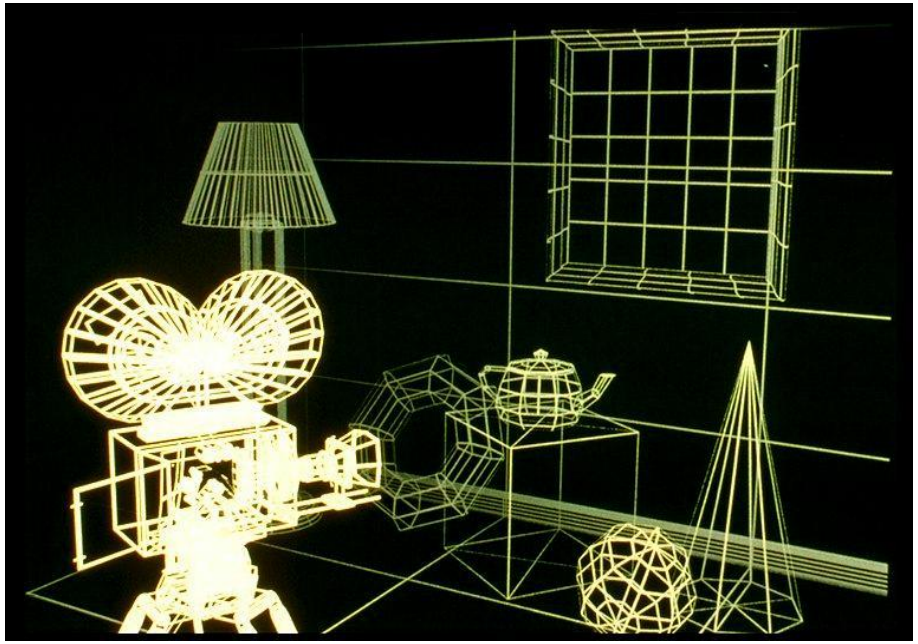


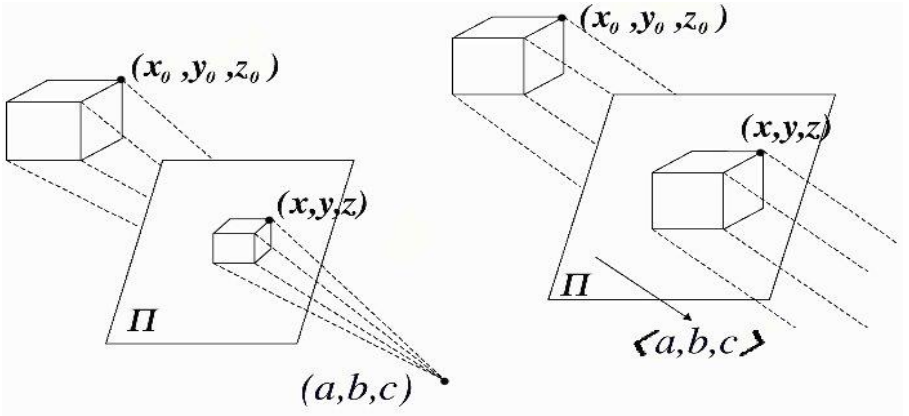
Perspective Projection

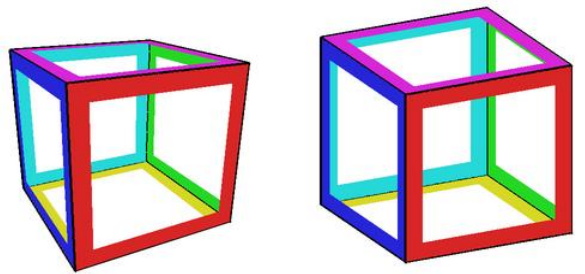
We want to create a picture of the scene viewed from the camera

We apply a perspective transformation to convert the 3D coordinates to 2D coordinates of the screen

Objects far away appear smaller, closer objects appear bigger

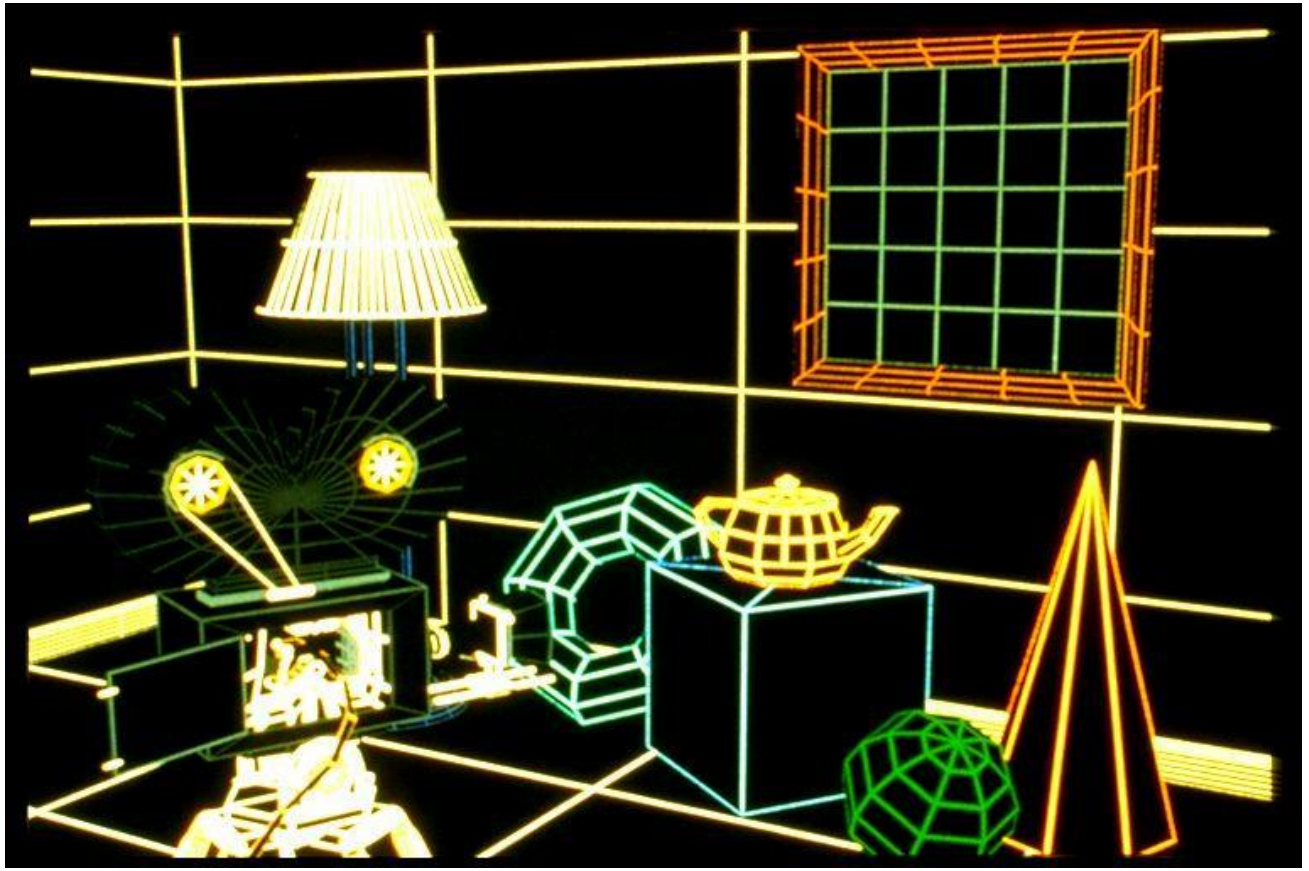






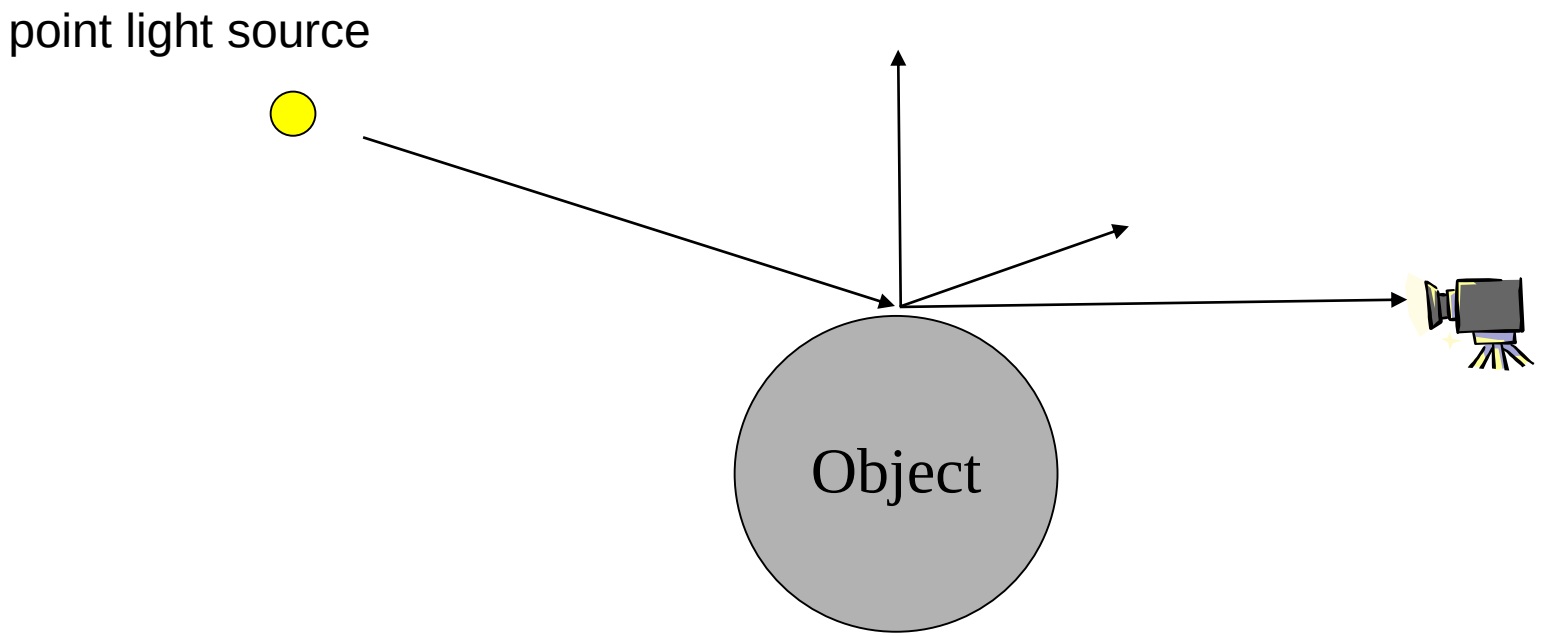
Hidden Surface Removal

Objects occluded by other objects must not be drawn



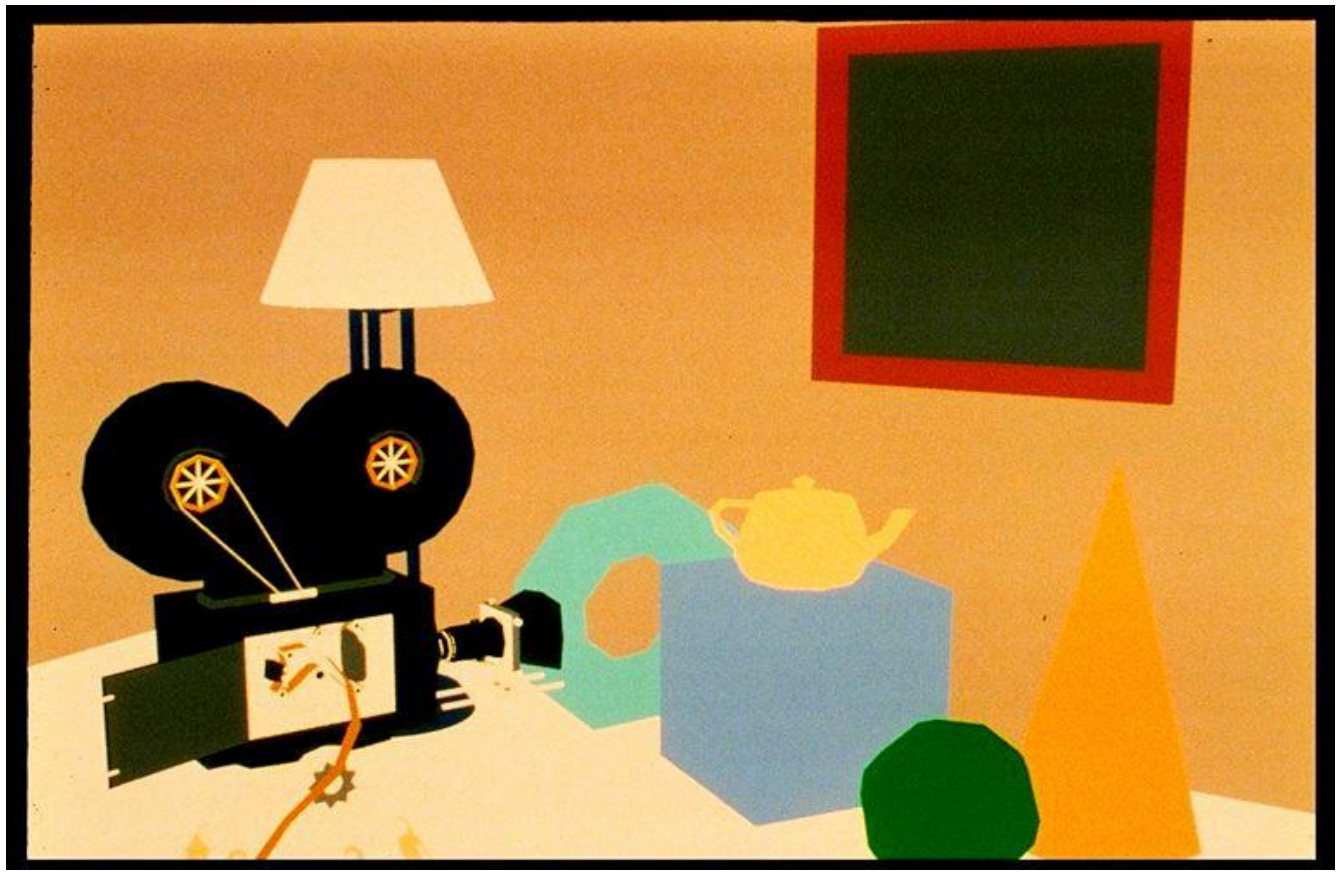
Shading

Now we need to decide the colour of each pixels taking into account the object's colour, lighting condition and the camera position



Shading : Constant Shading - Ambient

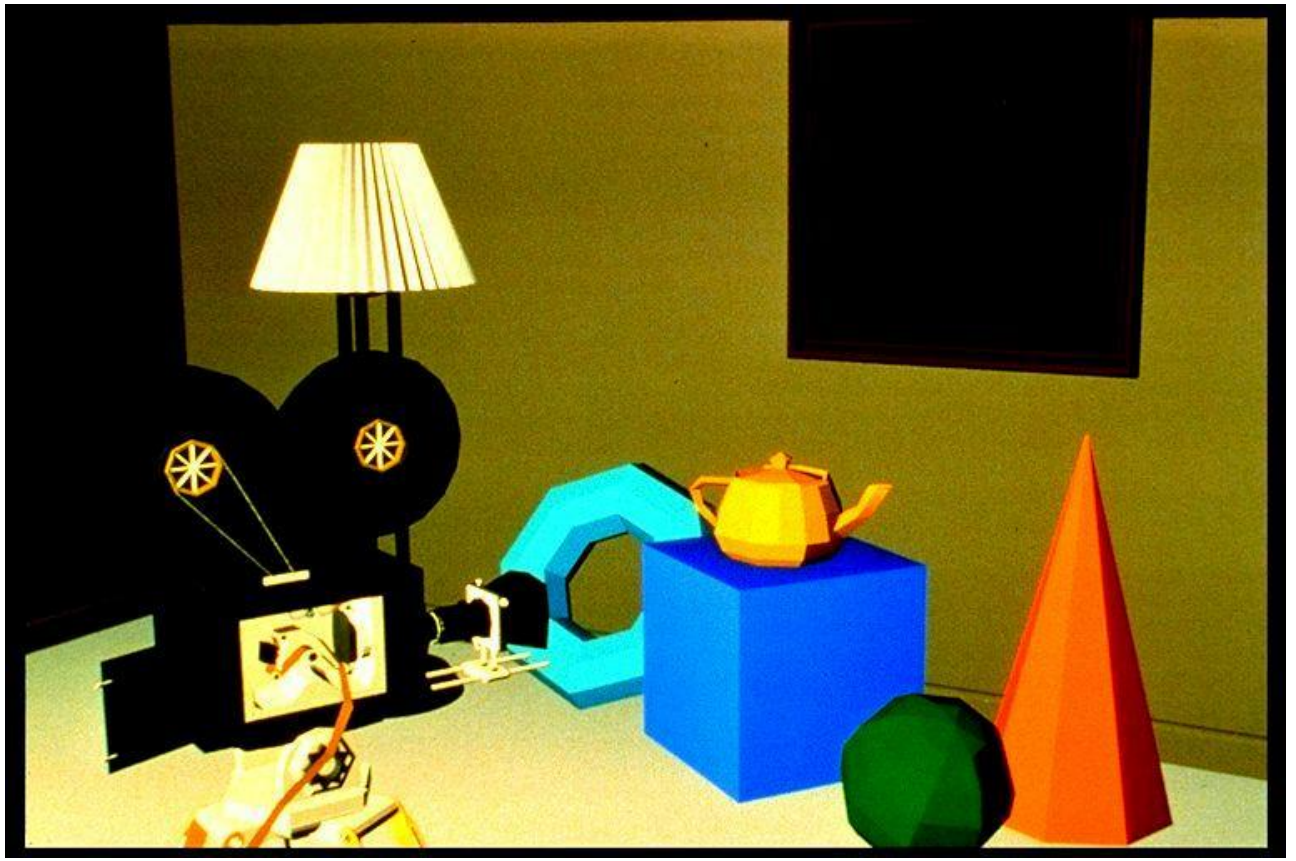
Objects colours by its own colour



Shading – Flat Shading

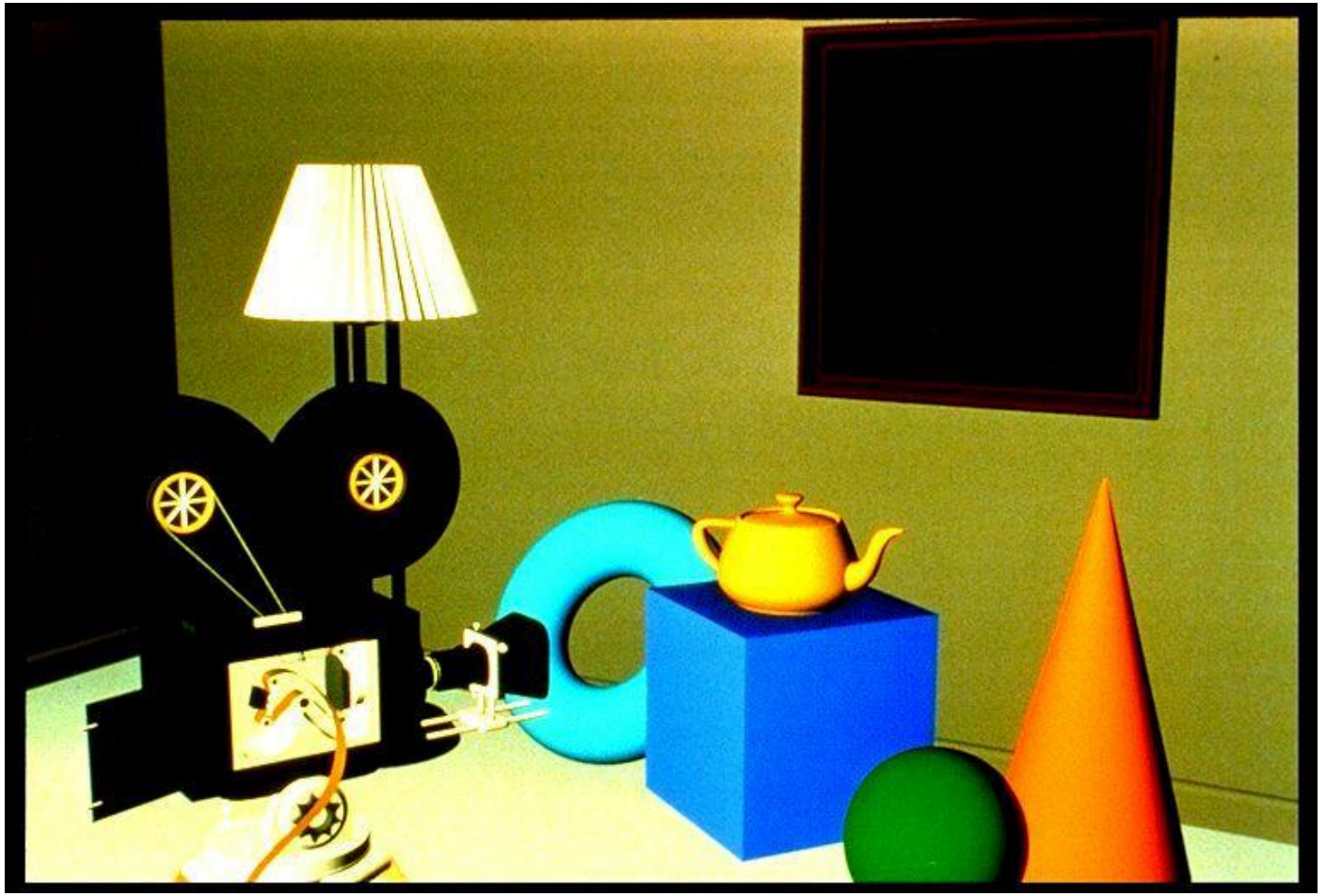
Objects colored based on its own colour and the lighting condition

One colour for one face

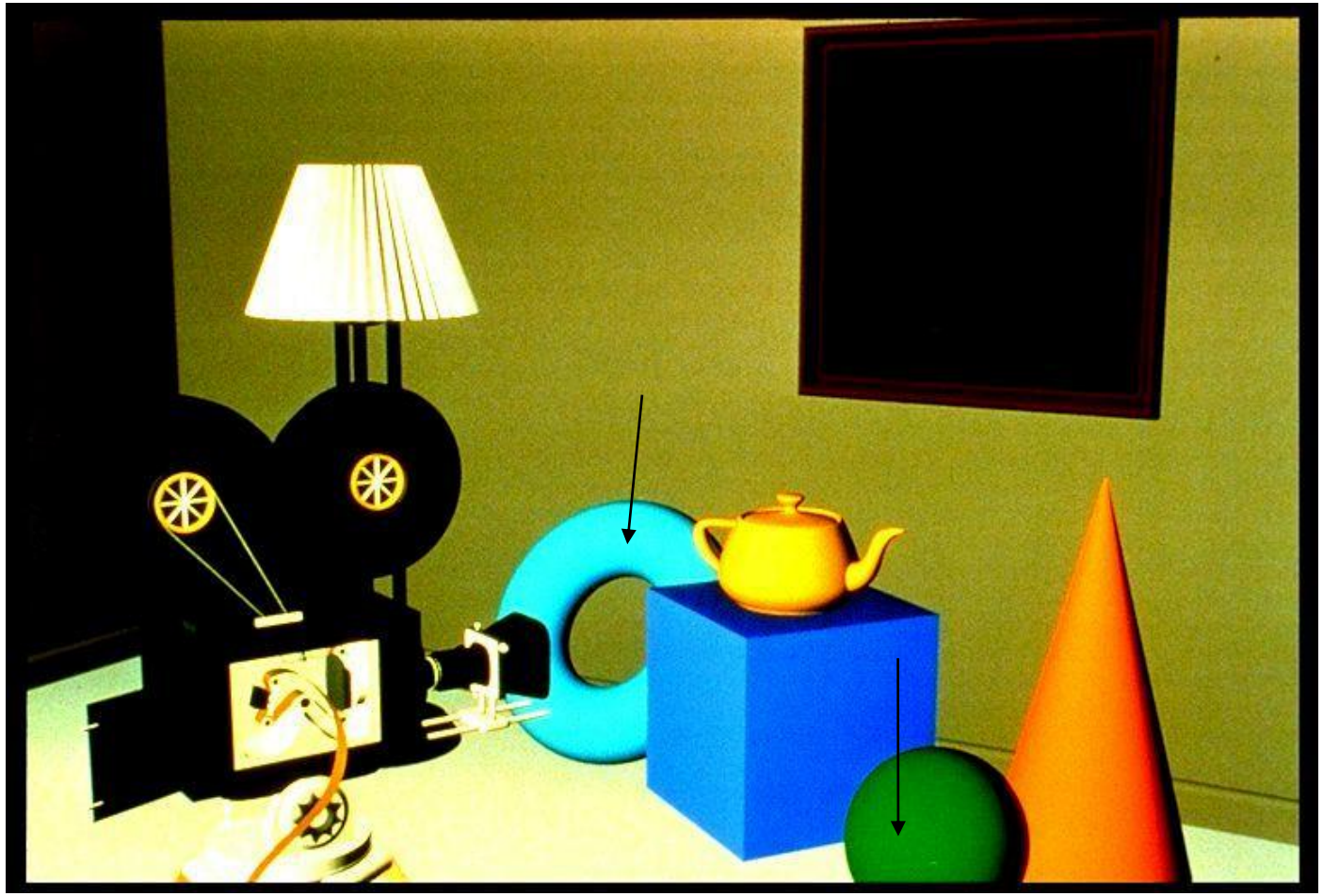


Gouraud shading, no specular highlights

Lighting calculation per vertex

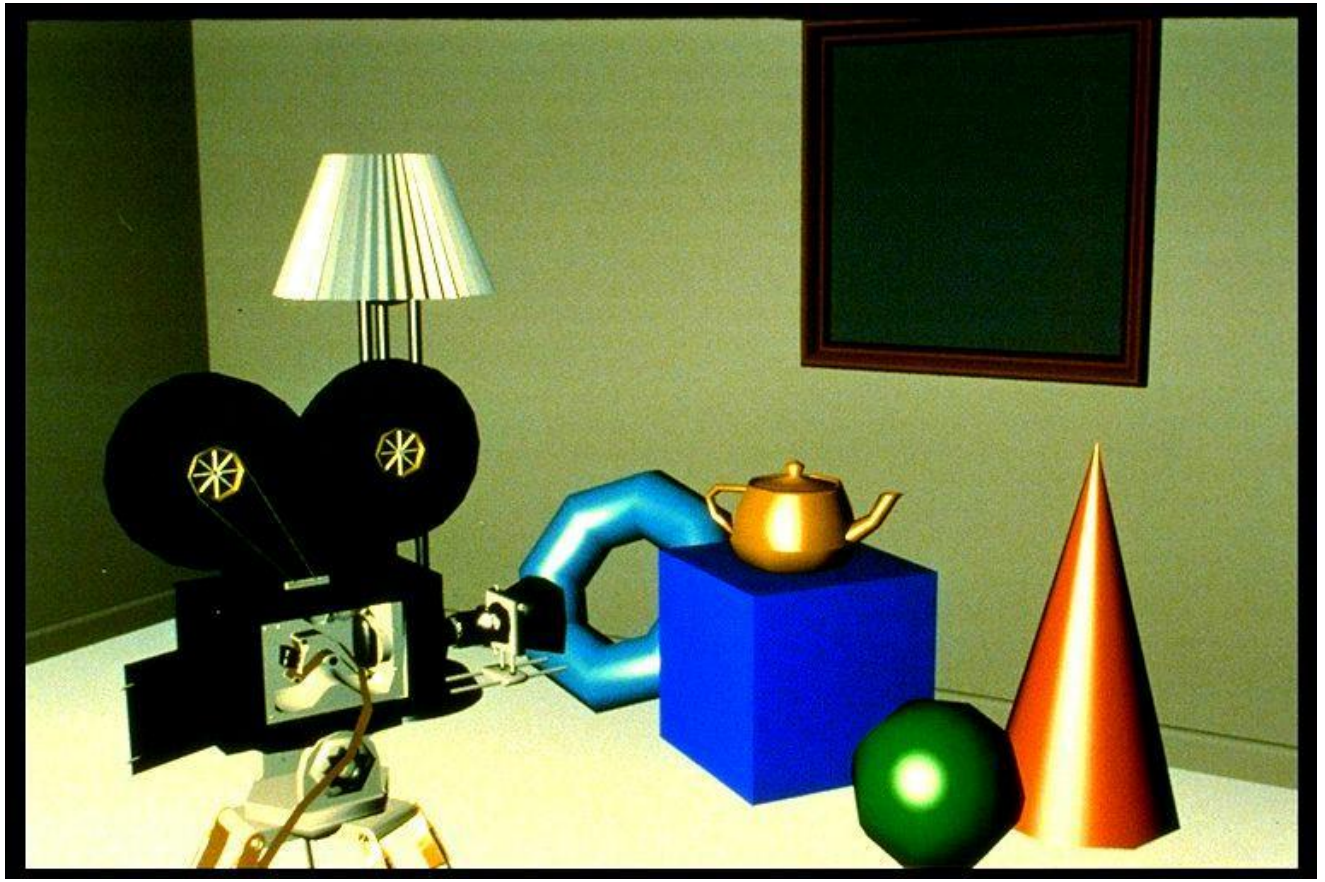


Shapes by Polynomial Surfaces

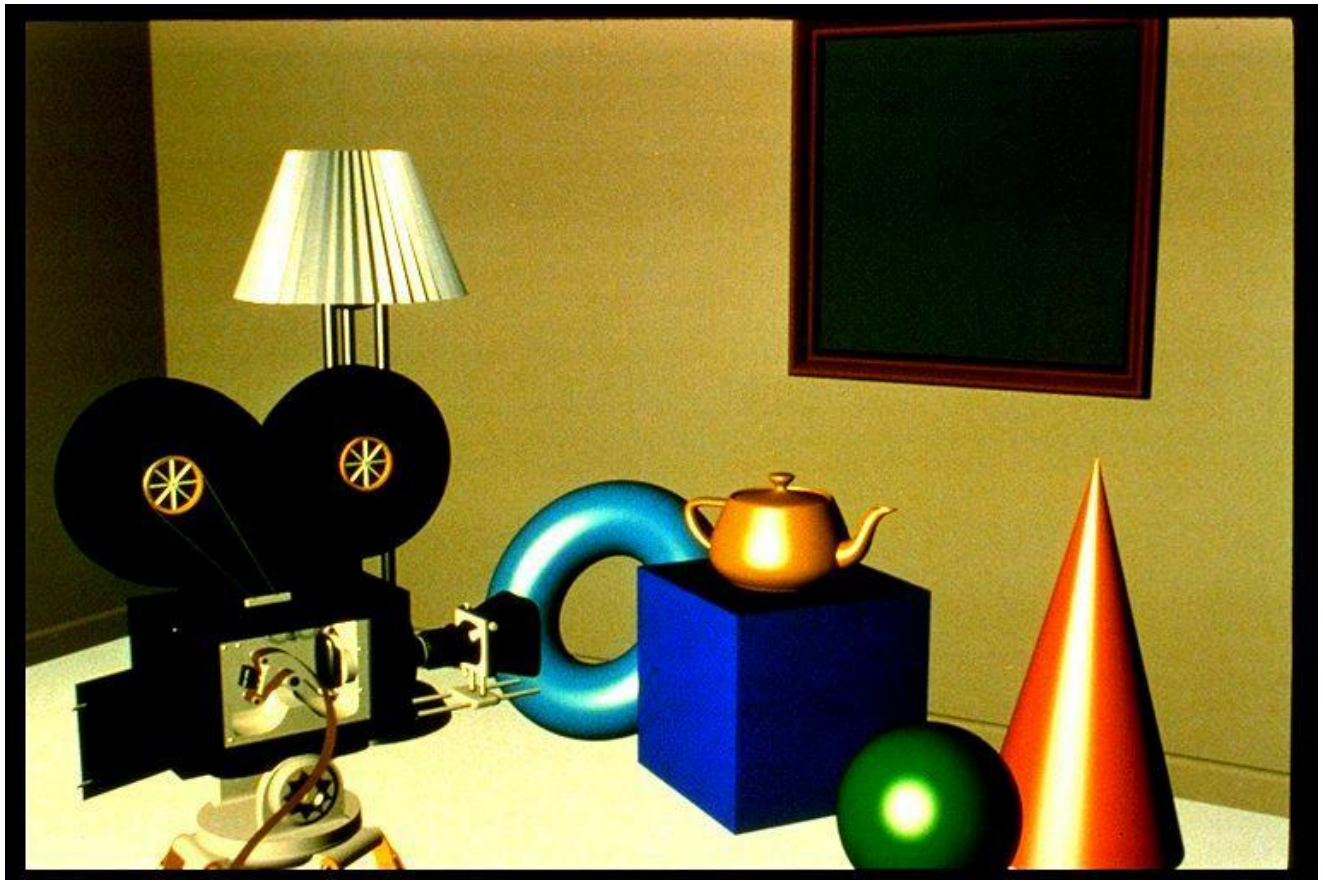


Specular highlights added

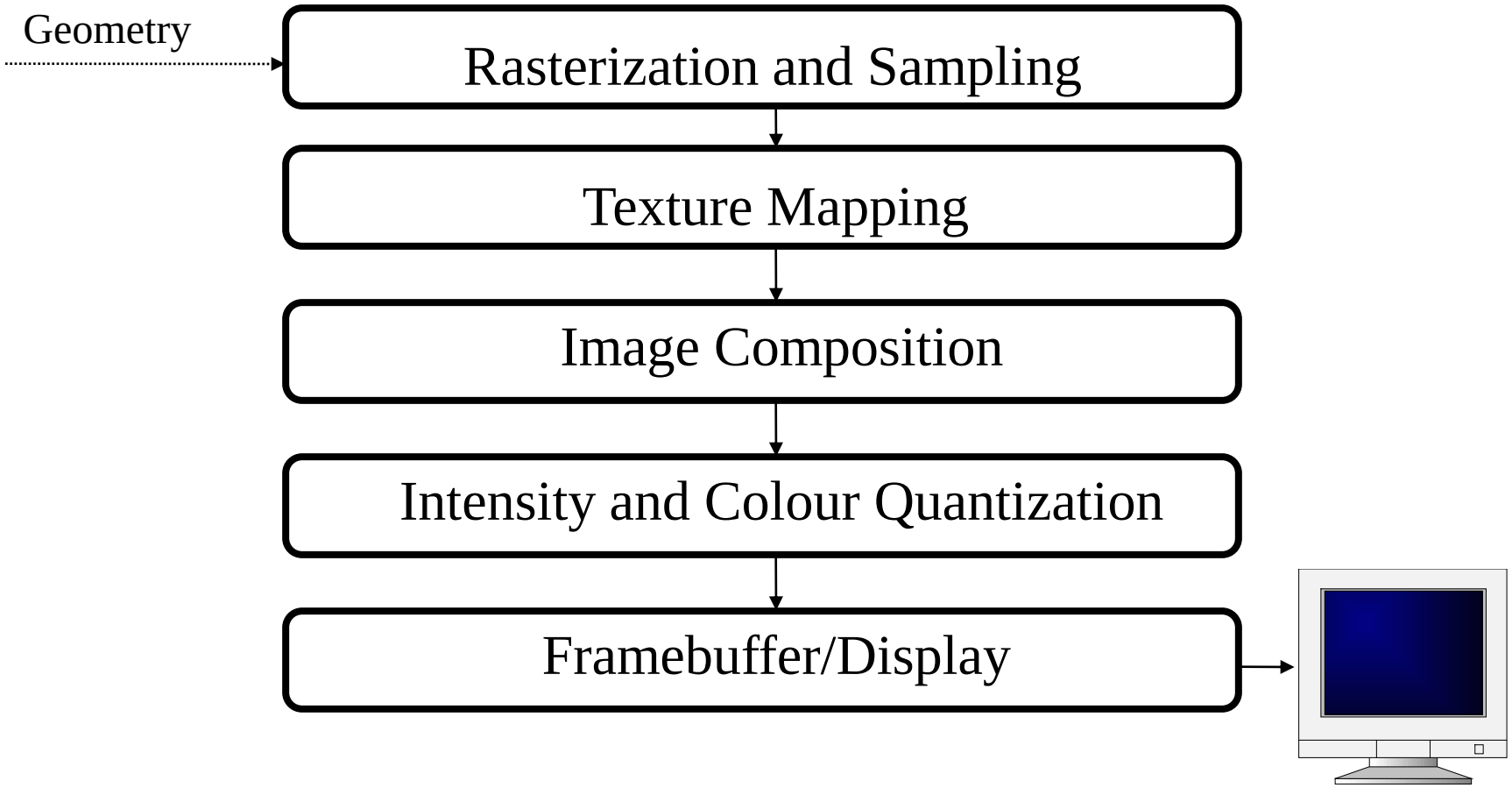
Light perfectly reflected in a mirror-like way



Phong shading



Next, the Imaging Pipeline

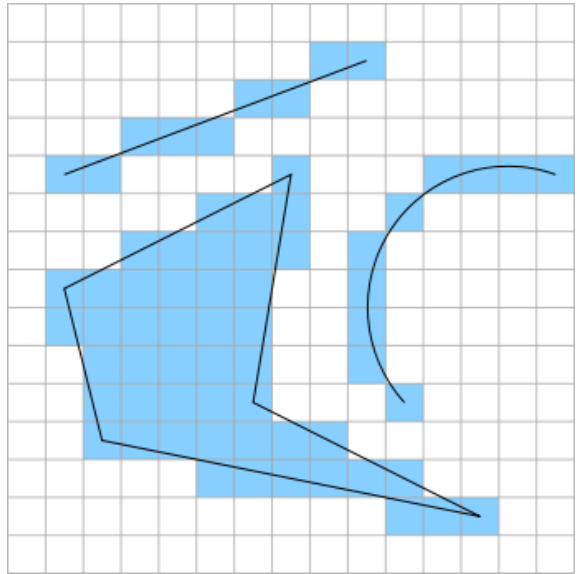


Rasterization

Converts the vertex information output by the geometry pipeline into pixel information needed by the video display

Aliasing: distortion artifacts produced when representing a high-resolution signal at a lower resolution.

Anti-aliasing : technique to remove aliasing

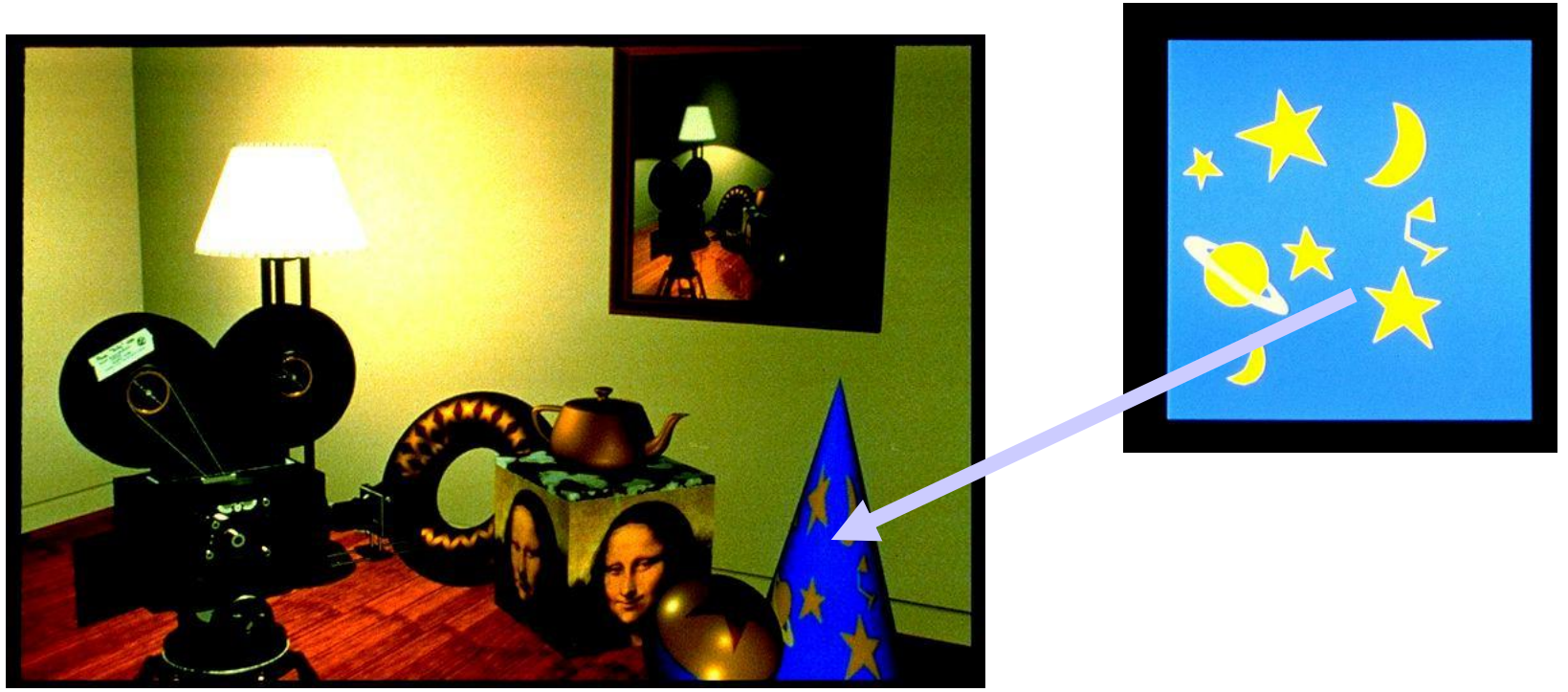


Anti-aliasing

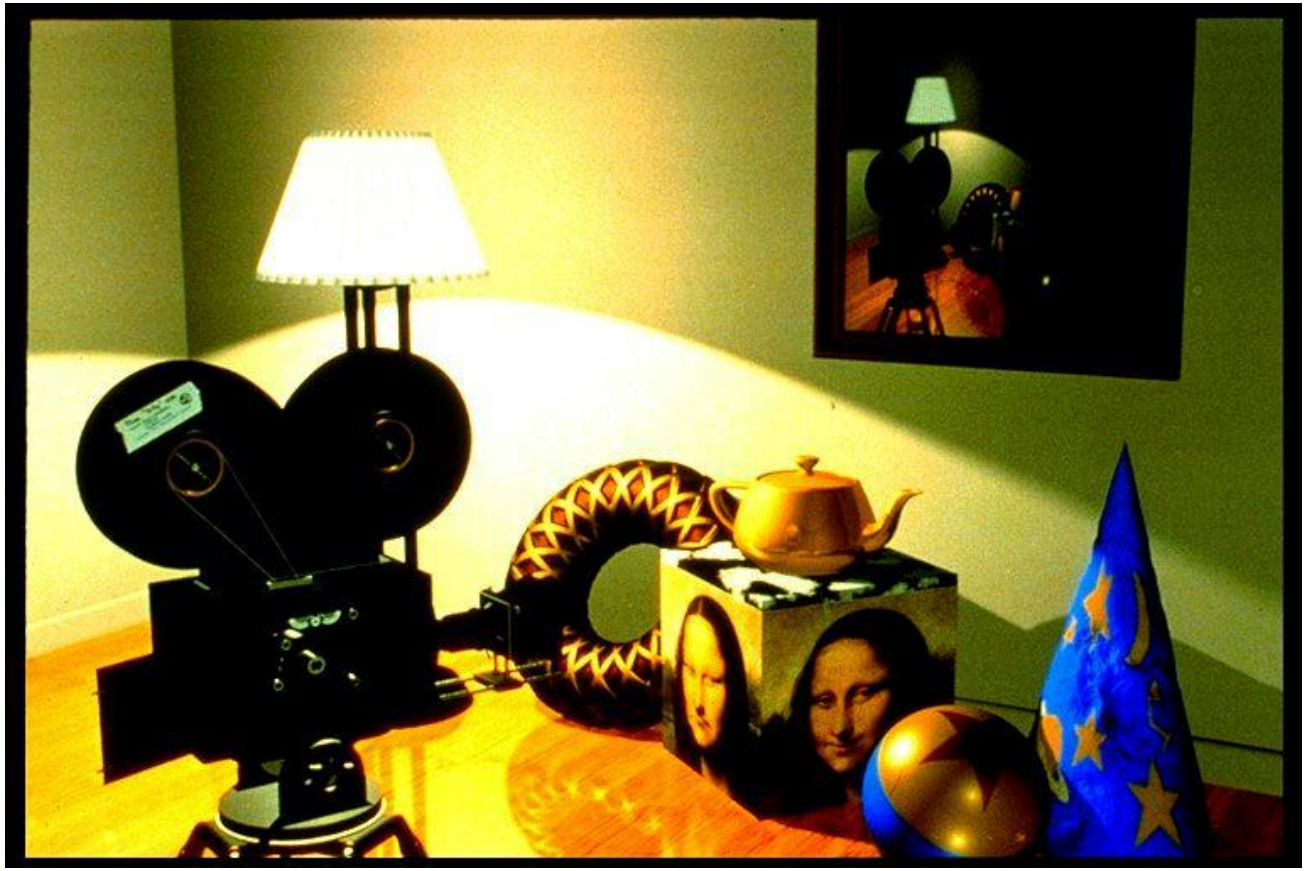
Aliased polygons (jagged edges)

Anti-aliased polygons

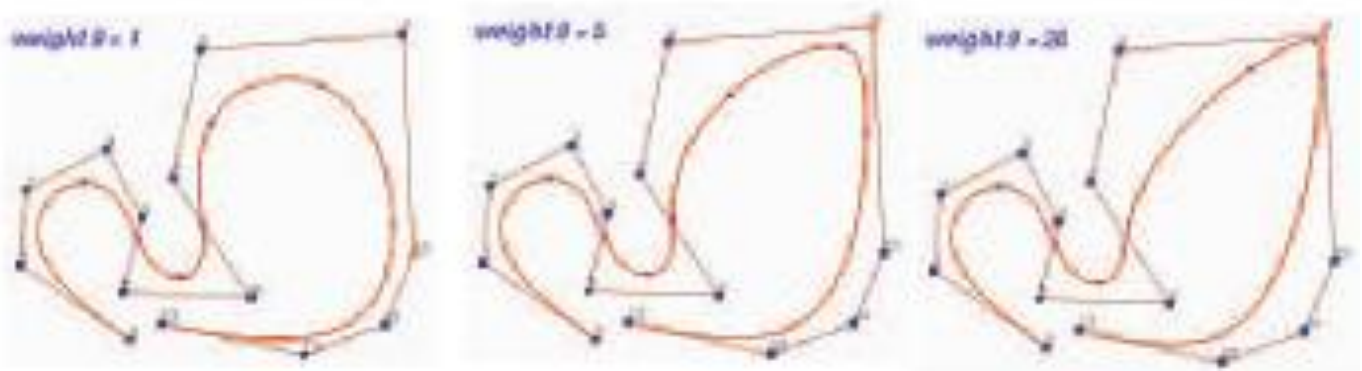
Texture mapping

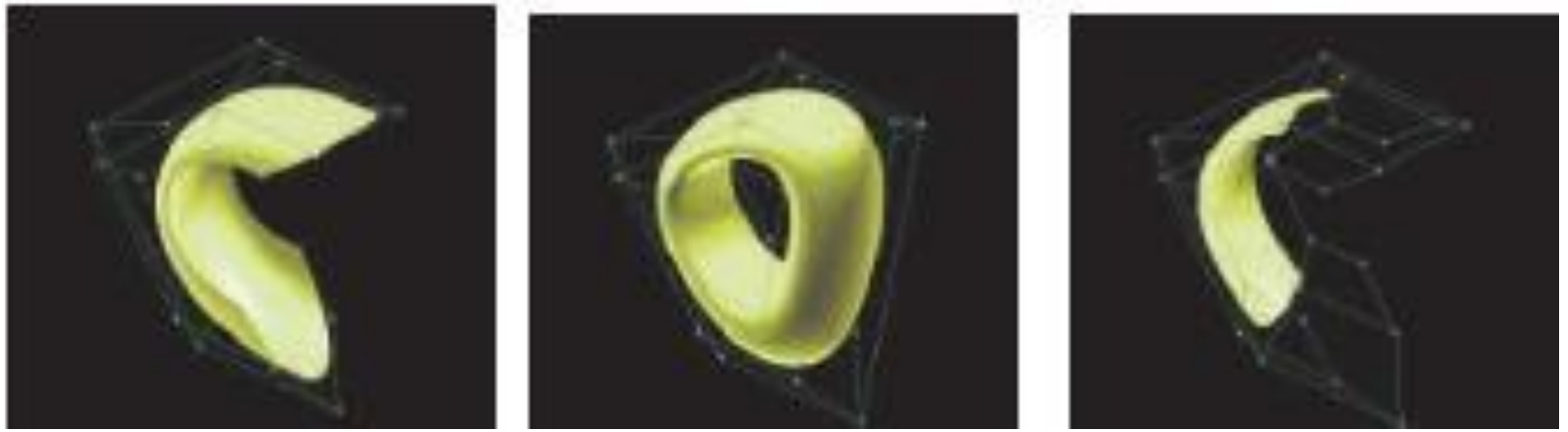


Other covered topics:
Reflections, shadows & Bump mapping



Polynomial Curves, Surfaces





Summary

The course is about algorithms, not applications

- Lots of mathematics

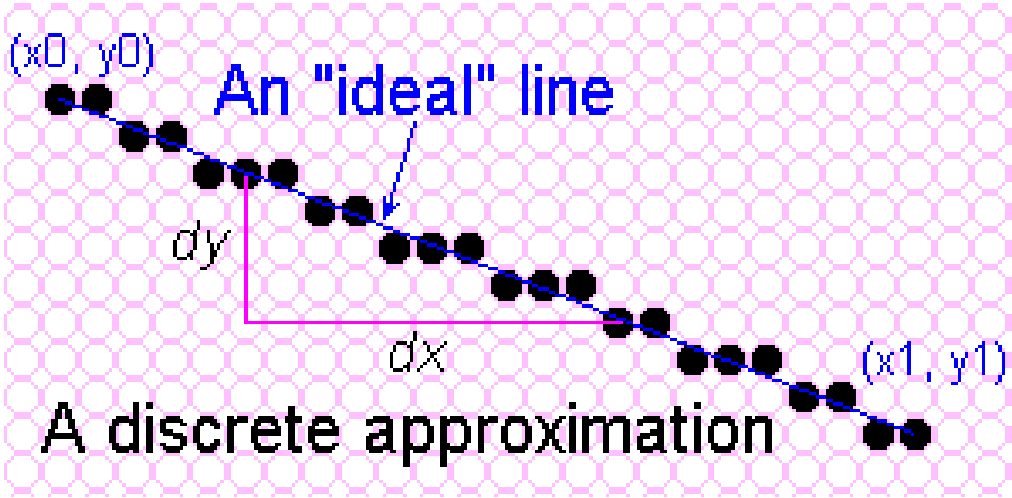
Graphics execution is a pipelined approach

Basic definitions presented

Some support resources indicated

Towards the Ideal Line

- We can only do a discrete approximation



- Illuminate pixels as close to the true path as possible, consider bi-level display only
  - Pixels are either lit or not lit

What is an *ideal* line

- Must appear straight and continuous (?)
  - Only possible axis-aligned and 45° lines
- Must interpolate both defining end points
- Must have uniform density and intensity
  - Consistent within a line and over all lines
  - What about antialiasing (?)
- Must be efficient, drawn quickly
  - Lots of them are required!!!

Simple Line

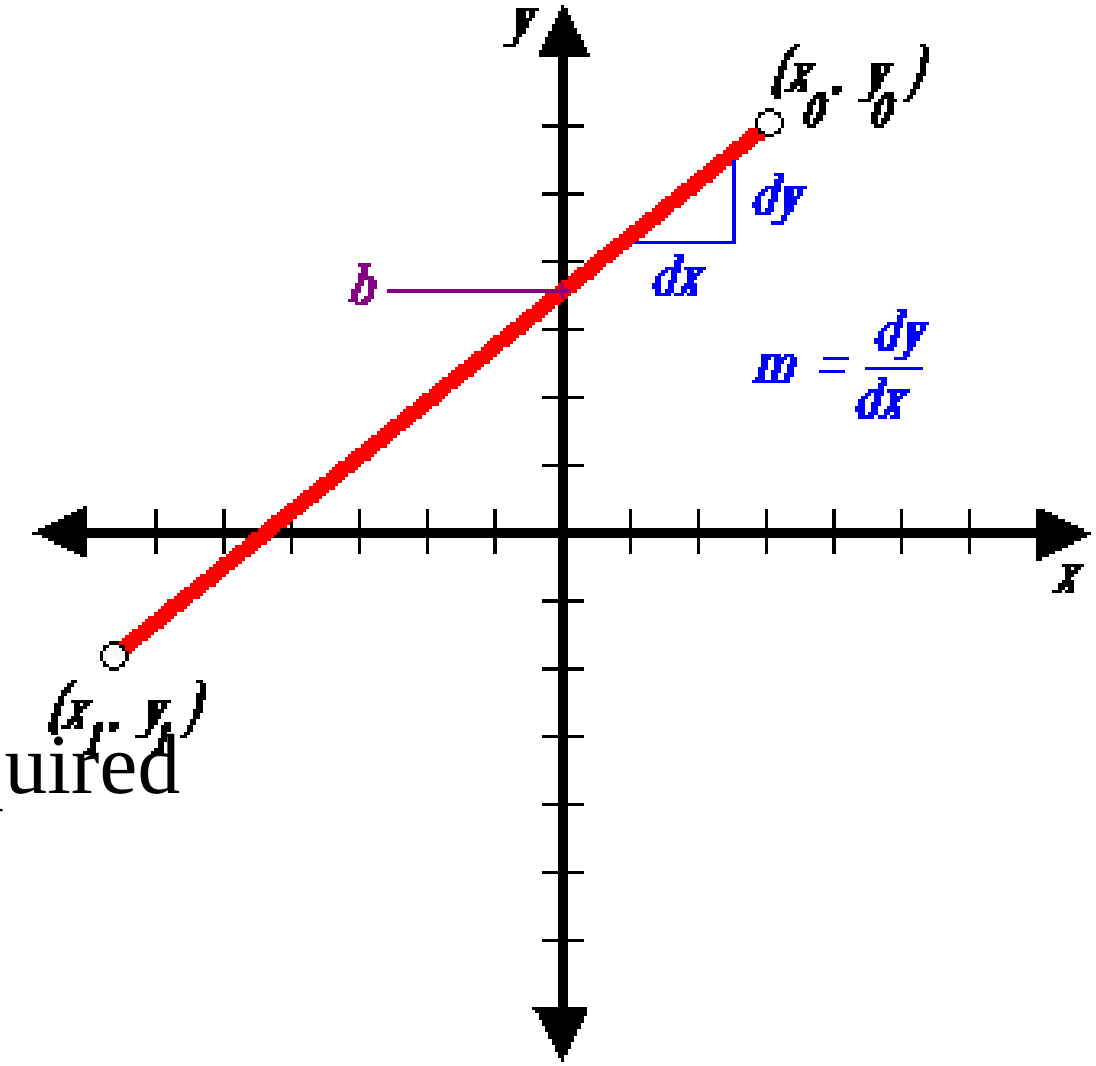
Based on *slope-intercept algorithm* from algebra:

$$y = mx + b$$

Simple approach:

increment x, solve for y

Floating point arithmetic required

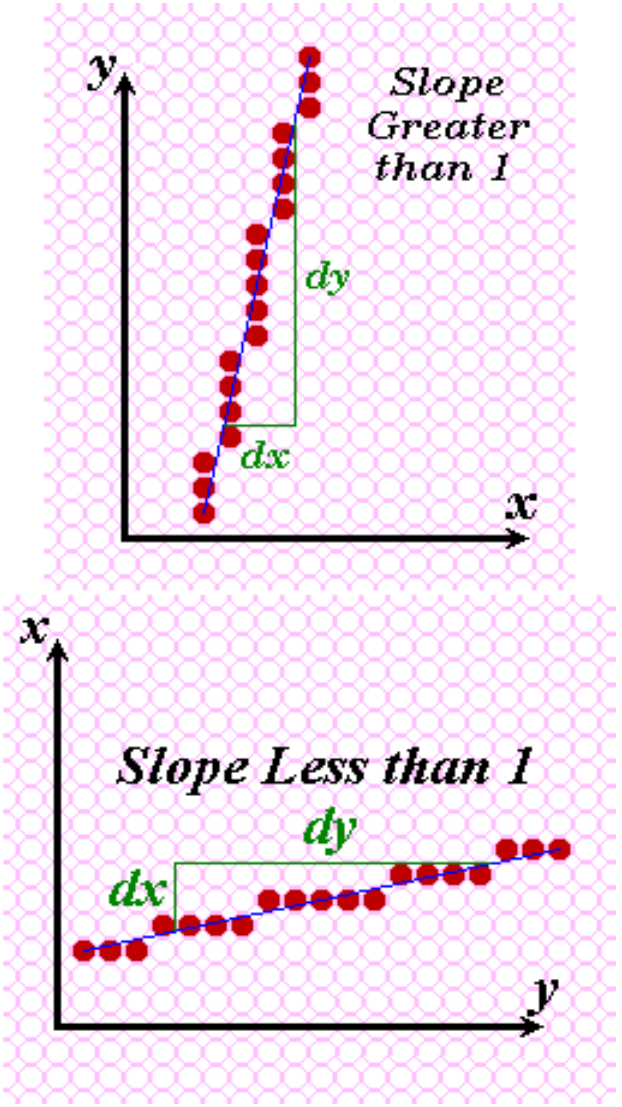


Does it Work?

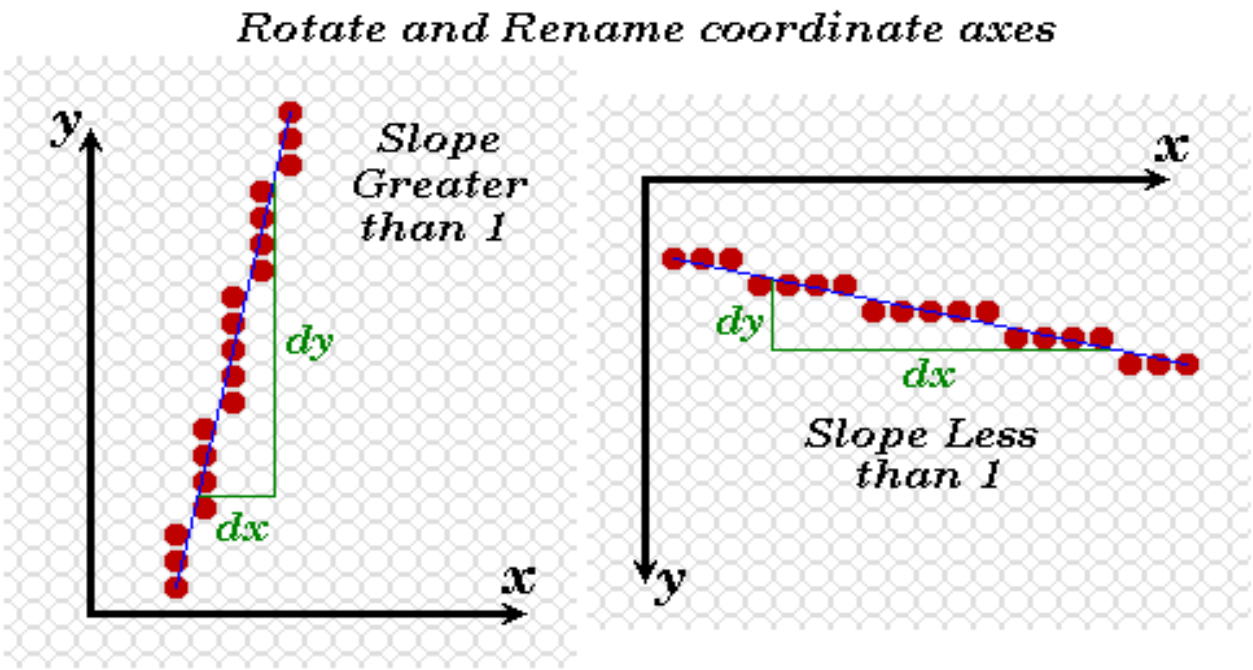
It seems to work okay for lines with a slope of 1 or less,

but doesn't work well for lines with slope greater than 1 – lines become more discontinuous in appearance and we must add more than 1 pixel per column to make it work.

Solution? - use *symmetry.*



Modify algorithm per octant



OR, increment along x-axis if dy<dx else increment along y-axis

DDA algorithm

- DDA = Digital Differential Analyser
  - finite differences
- Treat line as parametric equation in t :

Start point - (x_1, y_1)
End point - (x_2, y_2)

$$x(t) = x_1 + t(x_2 - x_1)$$
$$y(t) = y_1 + t(y_2 - y_1)$$

DDA Algorithm

$$x(t) = x_1 + t(x_2 - x_1)$$
$$y(t) = y_1 + t(y_2 - y_1)$$

- Start at $t = 0$
- At each step, increment t by dt

$$x_{new} = x_{old} + \frac{dx}{dt}$$
$$y_{new} = y_{old} + \frac{dy}{dt}$$

- Choose appropriate value for dt
- Ensure no pixels are missed:
  - Implies: and

$$dx = x_2 - x_1$$
$$dy = y_2 - y_1$$

$$\frac{dx}{dt} < 1 \qquad \frac{dy}{dt} < 1$$

- Set dt to maximum of dx and dy

DDA algorithm

```
line(int x1, int y1, int x2, int y2)

{
float x,y;
int dx = x2-x1, dy = y2-y1;
int n = max(abs(dx),abs(dy));
float dt = n, dxdt = dx/dt, dydt = dy/dt;
    x = x1;
    y = y1;
    while( n-- ) {
       point(round(x),round(y));
    x += dxdt;
    y += dydt;
    }
}
```

n - range of t.

DDA algorithm

- Still need a lot of floating point arithmetic.
  - 2 ‘round’s and 2 adds per pixel.

- Is there a simpler way ?
- Can we use only integer arithmetic ?
  - Easier to implement in hardware.

Summary of line drawing so far.

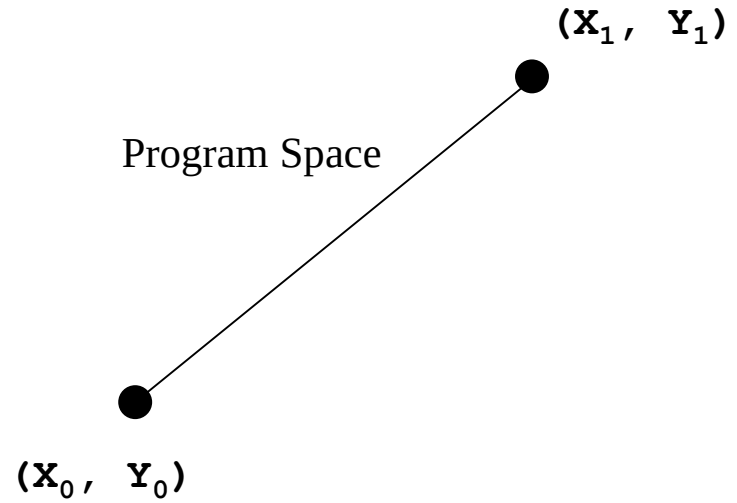
- Explicit form of line
  - Inefficient, difficult to control.
- Parametric form of line.
  - Express line in terms of parameter *t*
  - DDA algorithm
- Implicit form of line
  - Only need to test for ‘side’ of line.
  - Bresenham algorithm.
  - Can also draw circles.

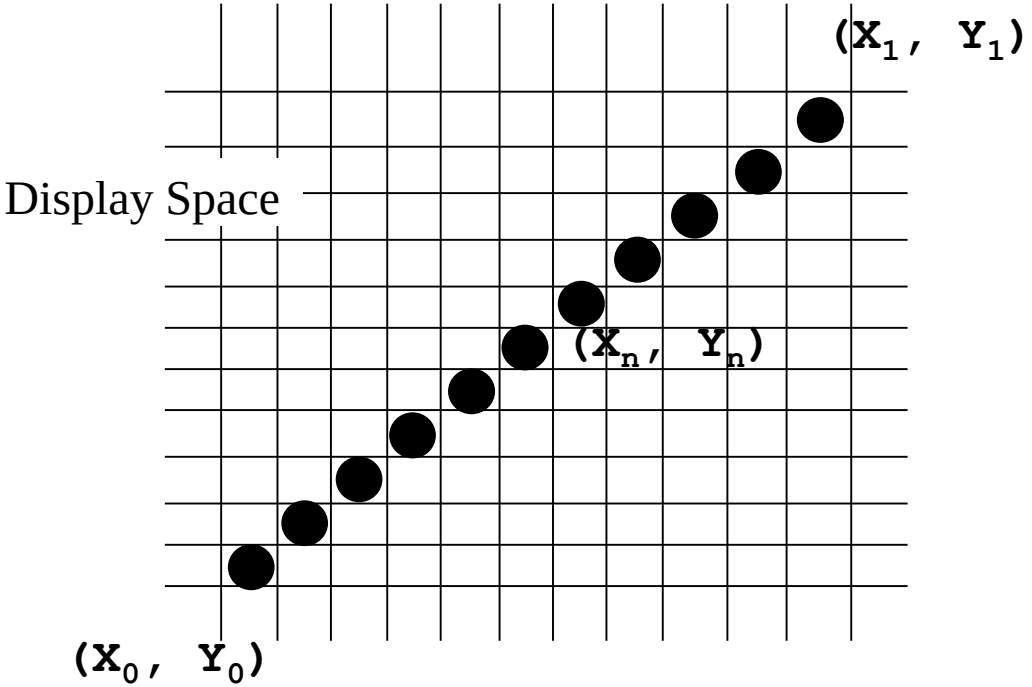
Scan Conversion

- Also known as *rasterization*
- In our programs objects are represented symbolically
  - 3D coordinates representing an object's position
  - Attributes representing color, motion, etc.
- But, the display device is a 2D array of pixels (unless you're doing holographic displays)
- Scan Conversion is the process in which an object's 2D symbolic representation is converted to pixels

Scan Conversion

- Consider a straight line in 2D
  - Our program will [most likely] represent it as two end points of a line segment which is not compatible with what the display expects

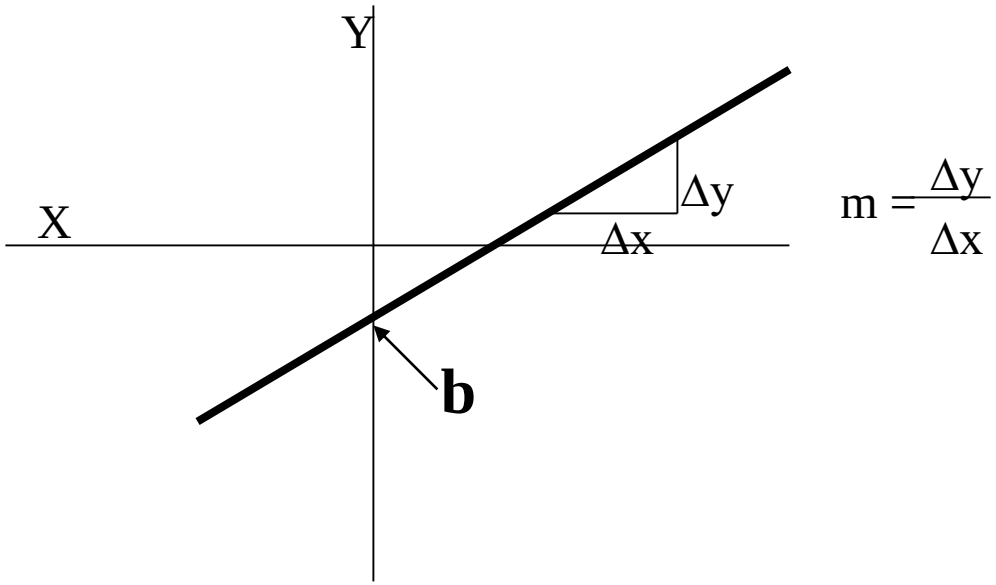




  - We need a way to perform the conversion

Line Drawing Algorithms

- Equations of a line
  - Point-Slope Form: $y = mx + b$ (Implicit form)
  - Also, $f(x,y) = Ax + By + C = 0$ (Explicit form)



$$m = \frac{\Delta y}{\Delta x}$$

As it turns out, this is not very convenient for scan converting a line

Drawing Lines

- Equations of a line
  - Two Point Form:

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

  - Directly related to the point-slope form but now it allows us to represent a line by two points – which is what most graphics systems use

Drawing Lines

- Equations of a line
  - Parametric Form:

$$x = x_0 + (x_1 - x_0)t$$

$$y = y_0 + (y_1 - y_0)t$$

  - By varying *t* from 0 to 1we can compute all of the points between the end points of the line segment – which is really what we want

Issues With Line Drawing

- Line drawing is such a fundamental algorithm that it must be done fast
  - Use of floating point calculations does not facilitate speed
- Furthermore, lines must be drawn without gaps
  - Gaps look bad, also create problem in continuity searching cases.
  - If you try to fill a polygon made of lines with gaps the fill will leak out into other portions of the display
  - Eliminating gaps through direct implementation of any of the standard line equations is difficult
- Finally, we don’t want to draw individual pixels more than once
  - That’s wasting valuable time

Midpoint/Bresenham's Line Drawing Algorithm

- Jack Bresenham addressed these issues with the *Bresenham Line Scan Convert* algorithm
  - This was back in 1965 in the days of *pen-plotters*
- All simple integer calculations
  - Addition, subtraction, multiplication by 2 (shifts)
- Guarantees connected (gap-less) lines where each point is drawn exactly 1 time
- Also known as the *midpoint line algorithm*

Bresenham's Line Algorithm

- Consider the two point-slope forms:

$$F(x,y) = Ax + By + C = 0 \qquad \text{Eqn. (1)}$$

$$y = \frac{\Delta y}{\Delta x} x + b$$

algebraic manipulation yields:

$$\Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot b = 0$$

where:

$$\Delta y = (y_1 - y_0); \Delta x = (x_1 - x_0)$$

yielding:

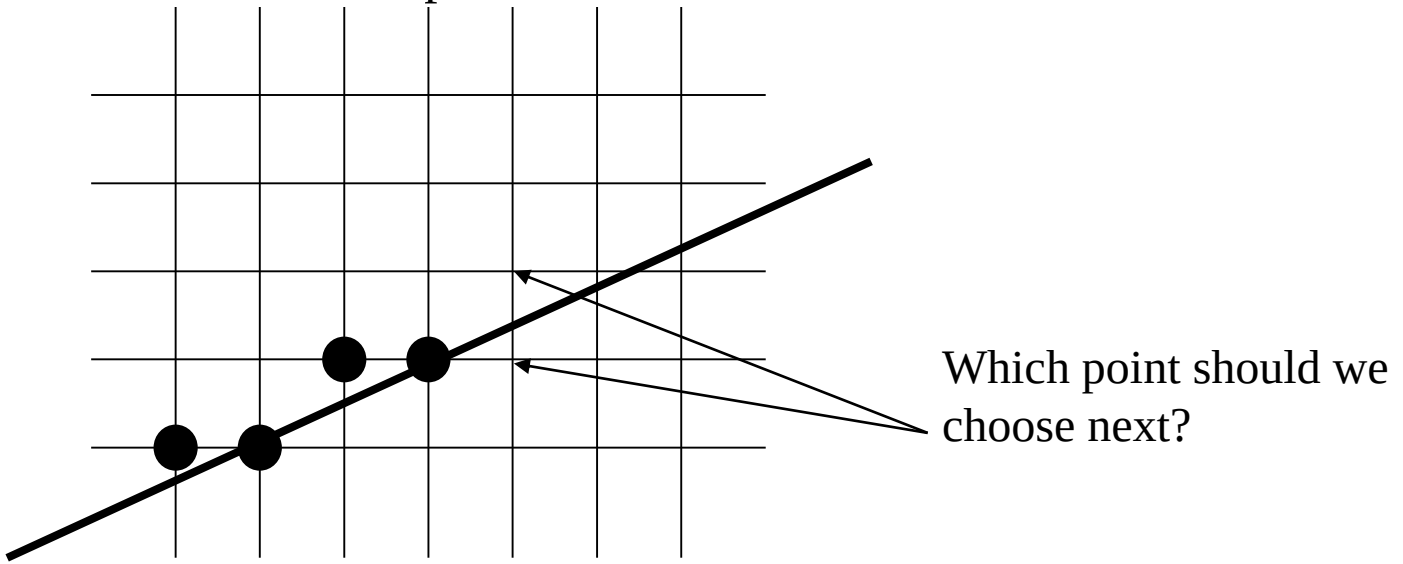
$$A = \Delta y; B = -\Delta x$$

Bresenham's Line Algorithm

- Assume that the slope of the line segment is

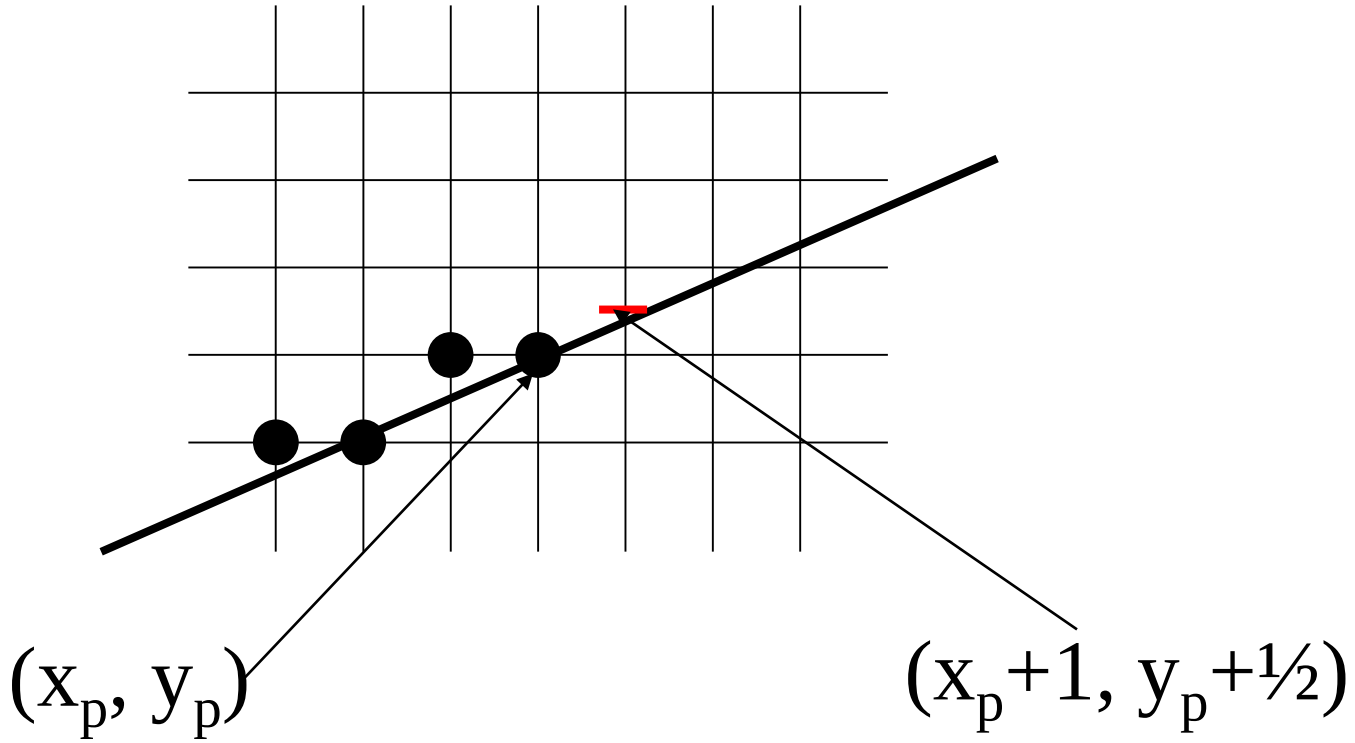
$$0 \leq m \leq 1$$

- Also, we know that our selection of points is restricted to the grid of display pixels
- The problem is now reduced to a decision of which grid point to draw at each step along the line
  - We have to determine how to make our steps



Bresenham's Line Algorithm

- What it comes down to is computing how close the midpoint (between the two grid points) is to the actual line



Bresenham's Line Algorithm

- Define $F(m) = d$ as the *discriminant or deviation*
  - (derive from the equation of a line $F(x,y) = Ax + By + C$)

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

$$F(m) = d_m = A \cdot (x_p + 1) + B \cdot (y_p + \frac{1}{2}) + C$$

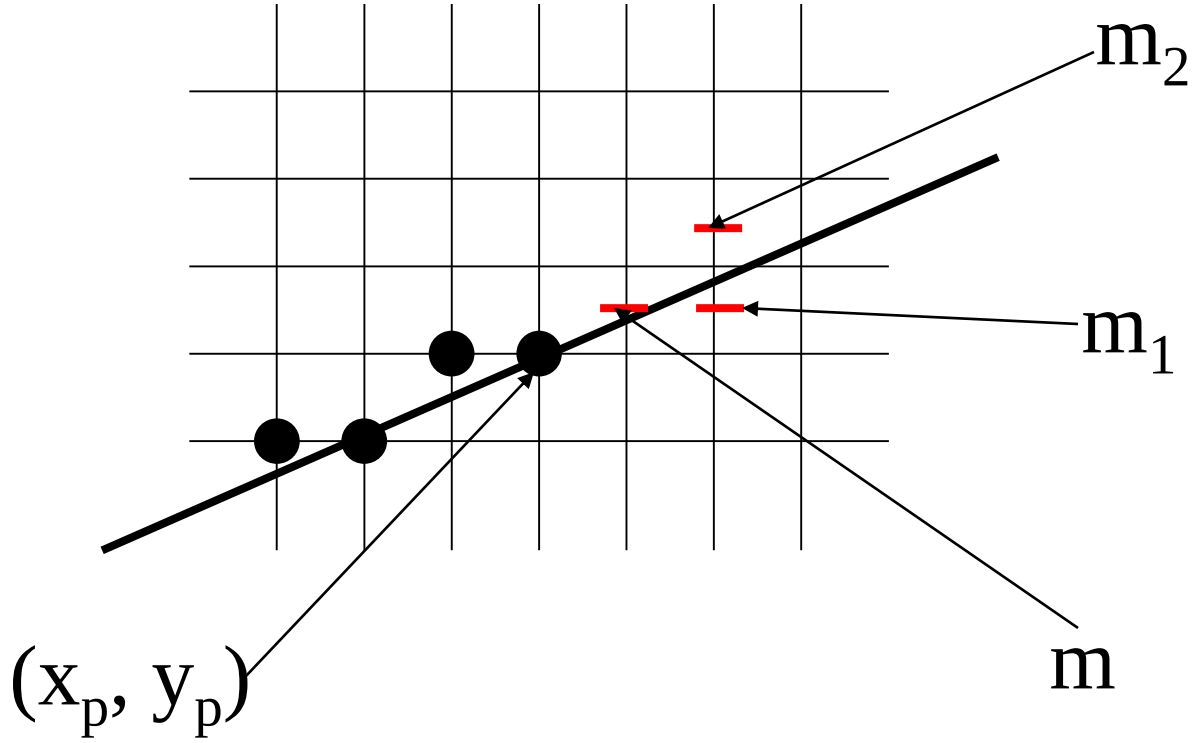
- if ($d_m > 0$) choose the upper point, otherwise choose the lower point

Bresenham's Line Algorithm

- But this is yet another relatively complicated computation for every point
- Bresenham's "trick" is to compute the discriminant *incrementally* rather than from scratch for every point

Bresenham's Line Algorithm

- Looking one point ahead we have:



Bresenham's Line Algorithm

- If d > 0 then discriminant is: (diagonal movement)

$$F(x_p+2, y_p+\frac{3}{2}) = d_{m2} = A\cdot(x_p+2)+B\cdot(y_p+\frac{3}{2})+C$$

- If d < 0 then the next discriminant is: (horizontal movement)

$$F(x_p+2, y_p+\frac{1}{2}) = d_{m1} = A\cdot(x_p+2)+B\cdot(y_p+\frac{1}{2})+C$$

- These can now be computed incrementally given the proper starting value

Bresenham's Line Algorithm

- Initial point is (x_0, y_0) and we know that it is on the line so $F(x_0, y_0)$ = must be 0, i.e., $Ax_0+By_0+ C = 0$
- Initial midpoint is $(x_0+1, y_0 + ½)$
- Initial discriminant is discriminant at $(x_0+1, y_0 + ½)$

$$F(x_0+1, y_0+½) = A(x_0+1) + B(y_0+½) + C$$
$$= (Ax_0+By_0+ C) + A + B/2$$
$$= F(x_0, y_0) + A + B/2$$

- And we know that $F(x_0, y_0) = 0$,
- So

$$d_{initial} = A + \frac{B}{2} = \Delta y - \frac{\Delta x}{2}$$

Bresenham's Line Algorithm

- Finally, to do this incrementally we need to know the differences between the current discriminant (d_m) and the two possible next discriminants (d_{m1} and d_{m2})

Bresenham's Line Algorithm

We know:

$$d_{current} = F(x_p + 1, y_p + \frac{1}{2}) = A \cdot (x_p + 1) + B \cdot (y_p + \frac{1}{2}) + C$$

$$d_{m1} = F(x_p + 2, y_p + \frac{1}{2}) = A \cdot (x_p + 2) + B \cdot (y_p + \frac{1}{2}) + C$$

$$d_{m2} = F(x_p + 2, y_p + \frac{3}{2}) = A \cdot (x_p + 2) + B \cdot (y_p + \frac{3}{2}) + C$$

We need (if we choose m1):

$$d_{m1} - d_{current} = A = (y_1 - y_0) = \Delta y = \Delta E$$

or (if we choose m2):

$$d_{m2} - d_{current} = A + B = (y_1 - y_0) - (x_1 - x_0) = \Delta y - \Delta x = \Delta NE$$

Bresenham's Line Algorithm

- Notice that ΔE and ΔNE both contain only constant, known values!

$$d_{m1} - d_{current} = A = (y_1 - y_0) = \Delta E$$

$$d_{m2} - d_{current} = A + B = (y_1 - y_0) - (x_1 - x_0) = \Delta NE$$

- Thus, we can use them incrementally – we need only add one of them to our current discriminant to get the next discriminant!

Bresenham's Line Algorithm

- The algorithm then loops through x values in the range of $x_0 \leq x \leq x_1$ computing a y for each x – then plotting the point (x, y)
- Update step
  - If the discriminant (*let* $d = d_{initial}$) is less than/equal to 0 then increment x only, leaving y alone, and increment the discriminant by ΔE ($d+=\Delta E$)
  - If the discriminant is greater than 0 then increment x, increment y, and increment the discriminant by ΔNE ($d+=\Delta NE$)
- This is for slopes less than or equal to 1
- If the slope is greater than 1 then loop on y (loop controller) and reverse the increments of *x's* and *y's*
- Sometimes you'll see implementations that the discriminant, ΔE, and ΔNE by 2 (to get rid of the floating point, initial divide by 2)
- And that is Bresenham's algorithm

Summary

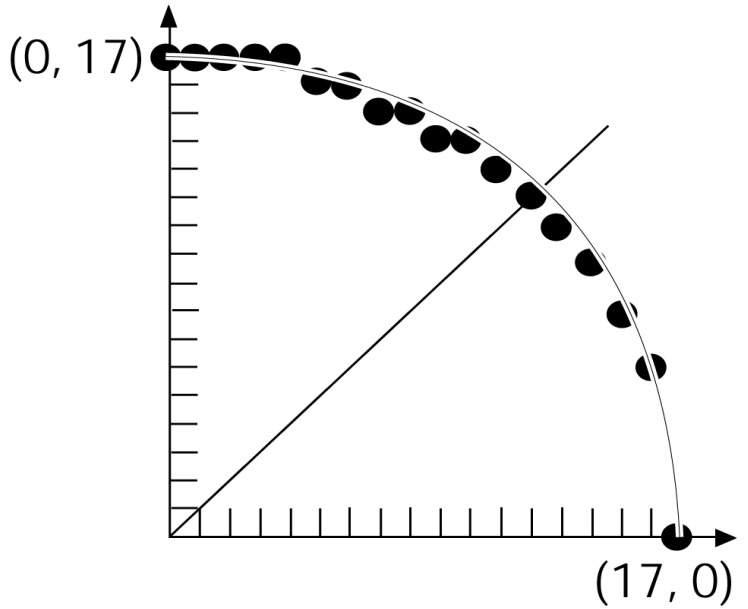
- Why did we go through all this?
- Because it's an extremely important algorithm
- Because the problem demonstrates the "need for speed" in computer graphics
- Because it relates mathematics to computer graphics (and the math is simple algebraic manipulations)
- Because it presents a nice programming assignment…

Implementation

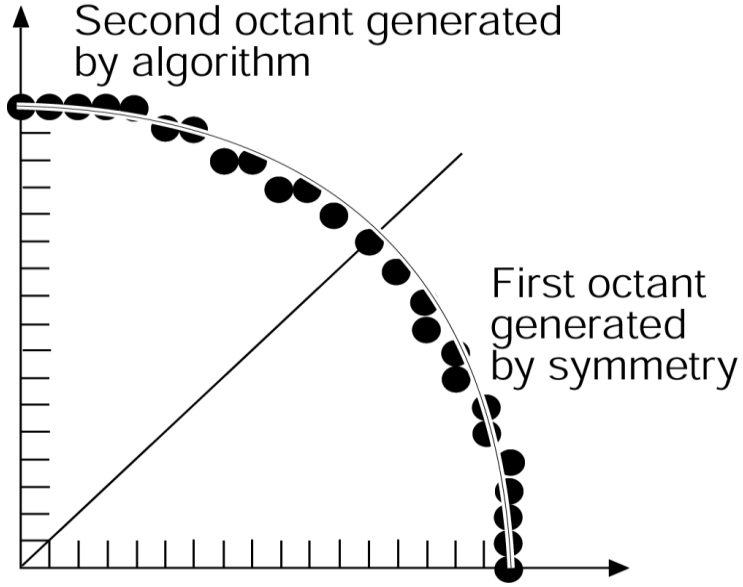
- Implement Bresenham's approach
  - You can search the web for this code – it’s out there
  - But, be careful because much of the code only does the slope < 1 case, so you’ll have to add the additional code so that the algorithm is applicable for any slope.

Scan Conversion of Circles

- Generalization of the line algorithm
- Assumptions:
  - circle at (0,0)
  - Fill 1/8 of the circle, then use 8-way symmetry



Not using the 8-way symmetry of a circle



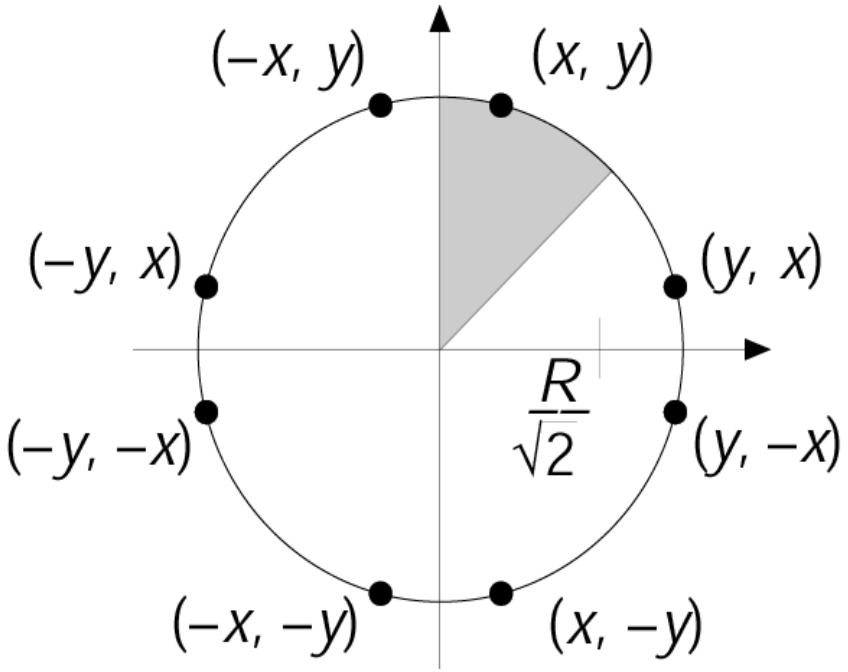
Using the 8-way symmetry of a circle:

Scan Conversion of Circles

- Implicit representation of the circle function:

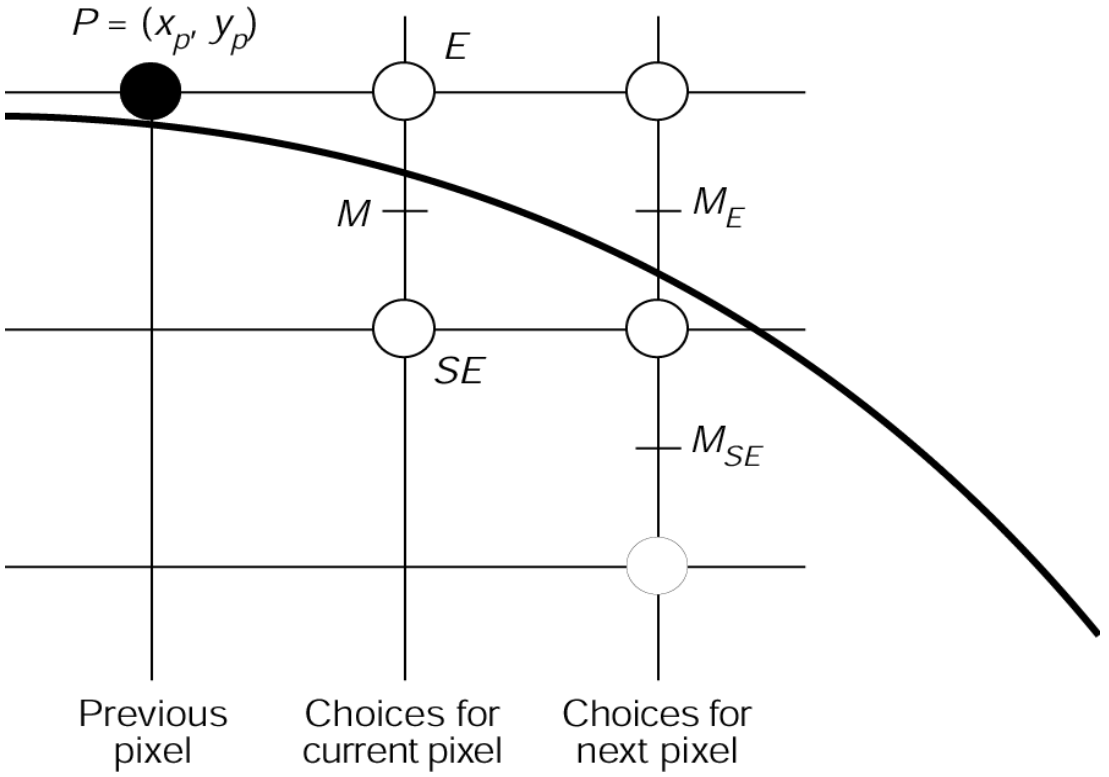
$$F(x,y) = x^2 + y^2 - R^2 = 0.$$

- Note: $F(x,y) < 0$ for points *inside* the circle, and $F(x,y) > 0$ for points *outside* the circle



Scan Conversion of Circles

- Assume we finished pixel $P = (x_p, y_p)$
- What pixel to draw next? (going clockwise)
- Note: the slope of the circular arc is between 0 and –1 (Zone-7)
  - Hence, choice is between:
    E and *SE*
- Idea:
  If the circle passes above the midpoint *M*, then we go to *E* next, otherwise we go to *SE*

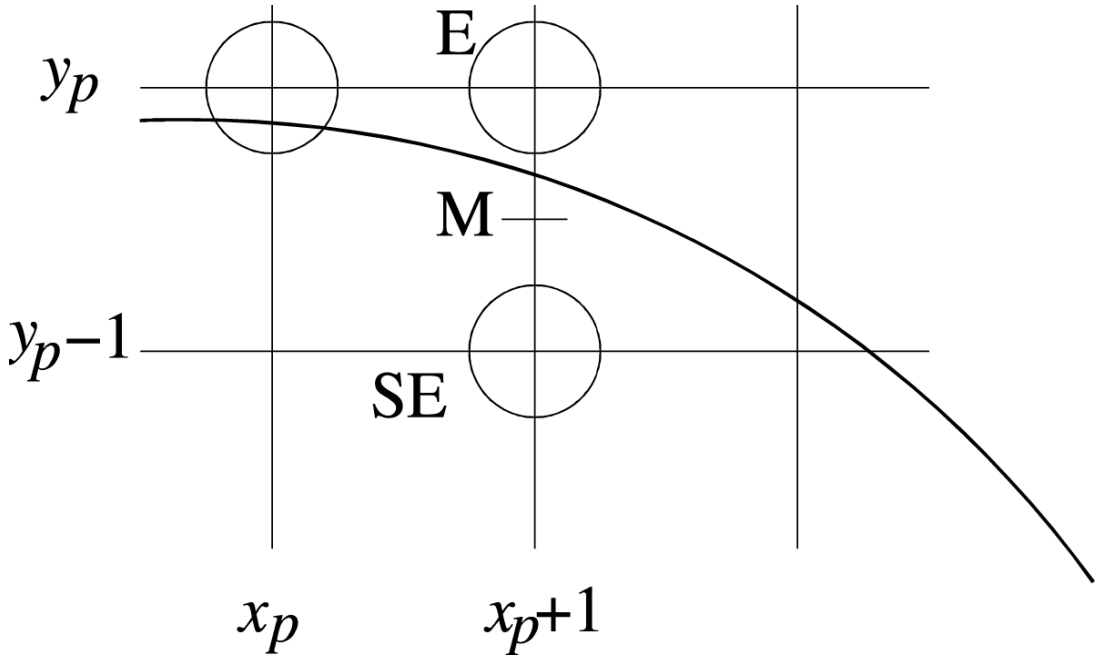


Scan Conversion of Circles

- We need discriminant D at midpoint M:

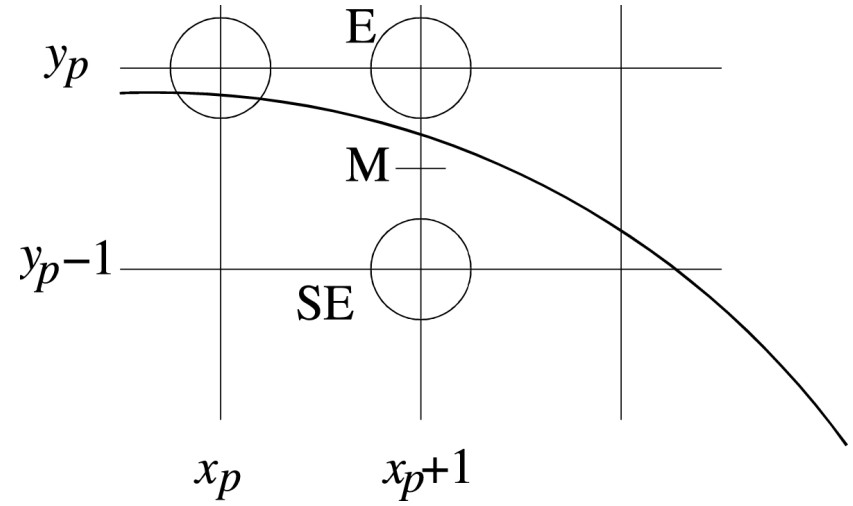
$$\begin{aligned} D &= F(M) = F(x_p + 1, y_p - \frac{1}{2}) \\ &= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2. \end{aligned}$$

- If $D < 0$ then M is *below* the arc, hence the *SE* pixel is closer to the line.
- If $D \geq 0$ then M is *above* the arc, hence the *E* pixel is closer to the line.



Case I: When *E* is next

- What increment for computing a new *D*?
- Next midpoint is: $(x_p + 2, y_p - (1/2))$

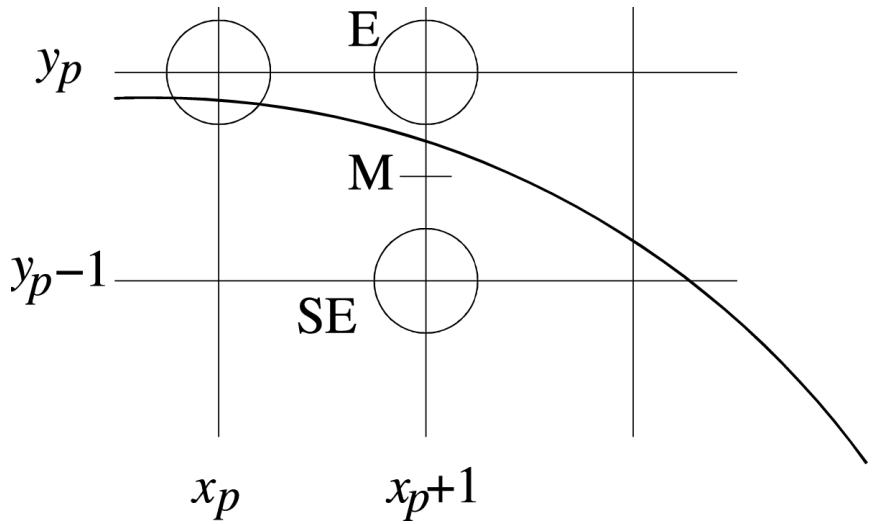


$$
\begin{aligned}
D_{new} &= F(x_p + 2, y_p - \frac{1}{2}) \\
&= (x_p + 2)^2 + (y_p - \frac{1}{2})^2 - R^2 \\
&= (x_p^2 + 4x_p + 4) + (y_p - \frac{1}{2})^2 - R^2 \\
&= (x_p^2 + 2x_p + 1) + (2x_p + 3) + (y_p - \frac{1}{2})^2 - R^2 \\
&= (x_p + 1)^2 + (2x_p + 3) + (y_p - \frac{1}{2})^2 - R^2 \\
&= D + (2x_p + 3).
\end{aligned}
$$

- Hence, increment by $\Delta E = (2x_p + 3)$

Case II: When *SE* is next

- What increment for computing a new *D*?
- Next midpoint is: $(x_p + 2,\ y_p - 1 - (1/2))$



$$
\begin{aligned}
D_{new} &= F(x_p + 2, y_p - \frac{3}{2}) \\
&= (x_p + 2)^2 + (y_p - \frac{3}{2})^2 - R^2 \\
&= (x_p^2 + 4x_p + 4) + (y_p^2 - 3y_p + \frac{9}{4}) - R^2 \\
&= (x_p^2 + 2x_p + 1) + (2x_p + 3) + (y_p^2 - y_p + \frac{1}{4}) + (-2y_p + \frac{8}{4}) - R^2 \\
&= (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2 + (2x_p + 3) + (-2y_p + 2) \\
&= D + (2x_p - 2y_p + 5)
\end{aligned}
$$

- Hence, increment by $\Delta SE = (2x_p - 2y_p + 5)$

Scan Conversion of Circles

- How to compute the *initial* value of D:
- We start with $x = 0$ and $y = R$, so the first midpoint is at

  $x = 1,\ y = R-1/2$:

$$
\begin{aligned}
D_{init} &= F(1, R - \frac{1}{2}) \\
&= 1 + (R - \frac{1}{2})^2 - R^2 \\
&= 1 + R^2 - R + \frac{1}{4} - R^2 \\
&= \frac{5}{4} - R.
\end{aligned}
$$

Scan Conversion of Circles

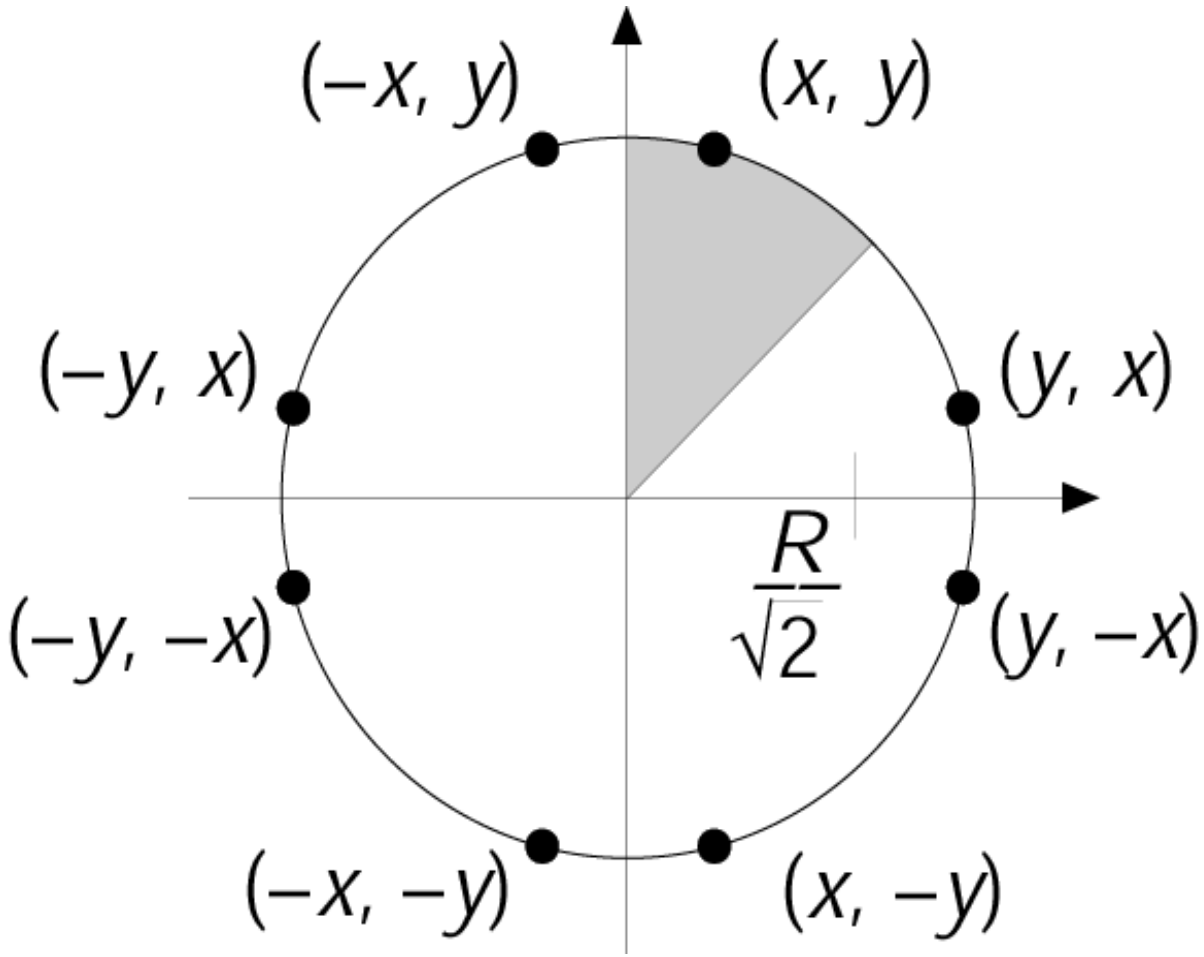
- Converting this to an integer algorithm:
  - Need only know if *D* is positive or negative
  - *D* & *R* are integers
  - Note D is incremented by an integer value
  - Therefore *D + 1/4* is positive only when *D* is positive; it is safe to drop the ¼ or Multiply 4 with all.
- Hence: set the initial *D* to *1 – R* (subtracting *1/4*) or *5 – 4R*

Scan Conversion of Circles

- Given radius R and center (0, 0)
  - First point (0, R)
- Initial decision parameter $D = 5 - 4R$
- While $x \leq y$
  - If ($D < 0$)
    - $x{+}{+}; D \mathrel{+}= 4(2x + 3);$
  - else
    - $x{+}{+}; y{-}{-}; D \mathrel{+}= 4(2(x - y) + 5);$
  - CirclePixel(x, y)

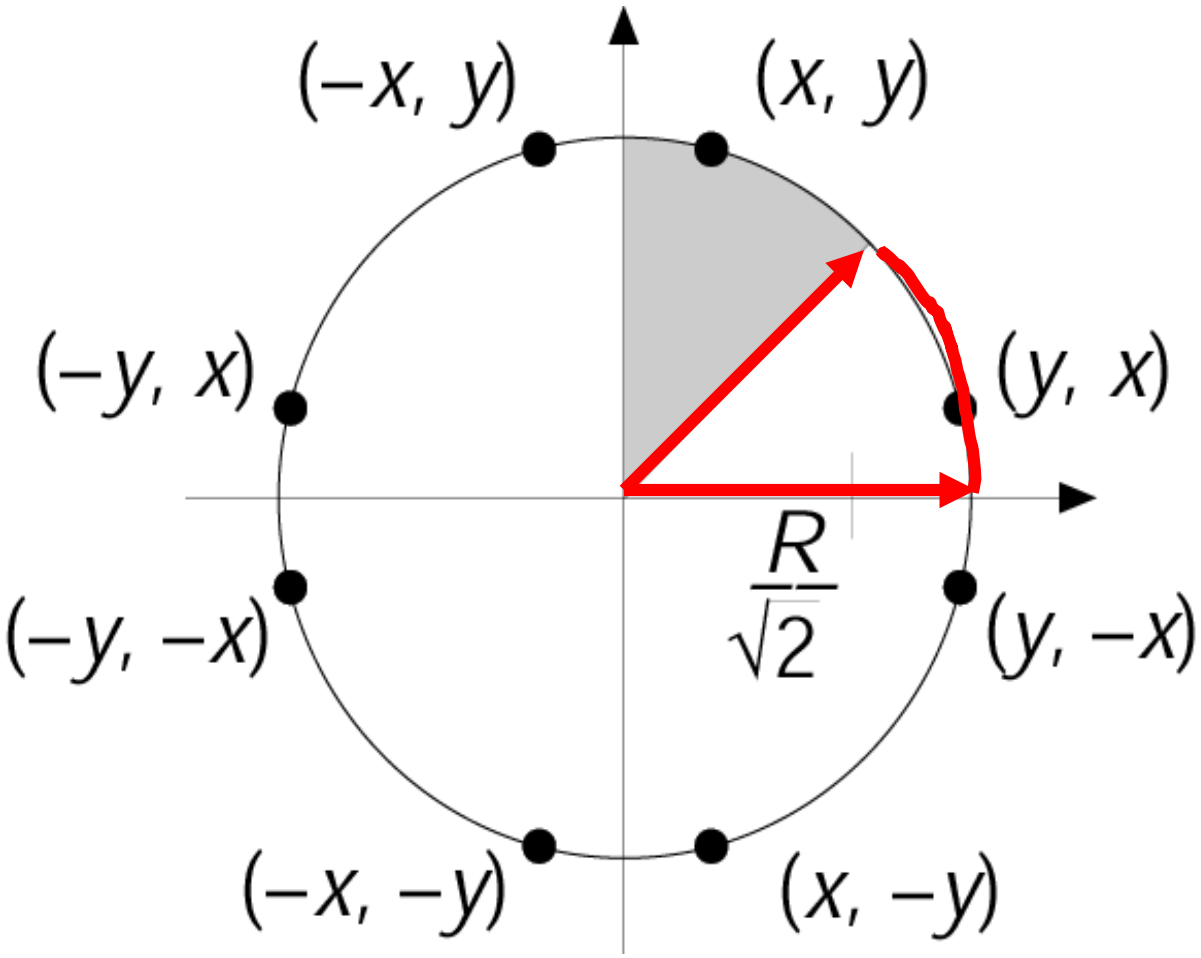
CirclePixel(*x*, *y*)

- Writes pixels to the seven other octants



Next Pixel Calculation for other Octant

- Derive D_{init} and the rate of change of discriminants ΔNW and ΔN for the 1st octant (assuming counter-clockwise next pixel calculation).



Ellipse Drawing

Properties of Ellipses

- Equation simplified if ellipse axis parallel to coordinate axis

$$\left(\frac{x-x_c}{r_x}\right)^2+\left(\frac{y-y_c}{r_y}\right)^2=1$$

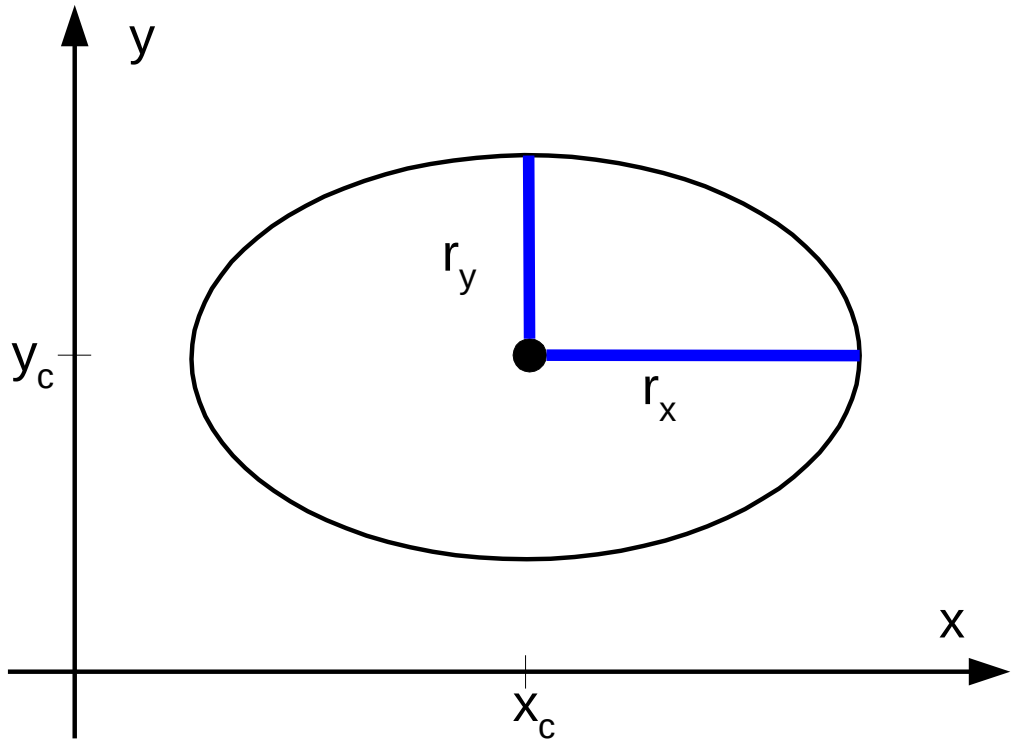
- If center of the ellipse is at origin,

$$f(x,y)=r_y^2x^2+r_x^2y^2-r_x^2r_y^{\,2}$$

- Parametric form

$$x=x_c+r_x\cos\theta$$

$$y=y_c+r_y\sin\theta$$



Symmetry Considerations

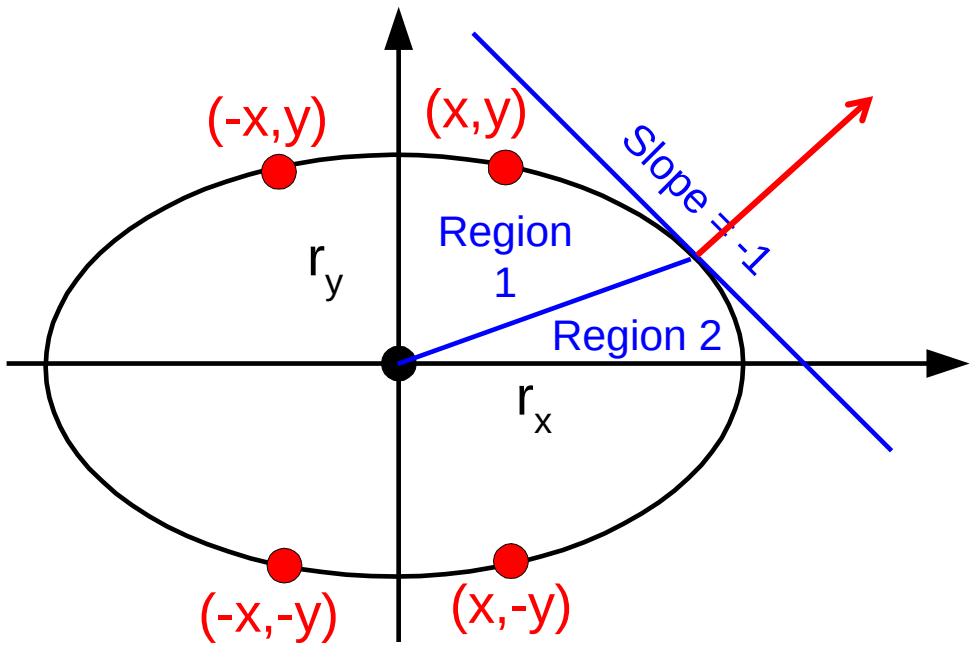
- 4-way symmetry
- Unit steps in *x* until reach region boundary
- Switch to unit steps in *y*

$$f(x,y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$

$$\frac{dy}{dx} = -\frac{r_y^2 x}{r_x^2 y}$$

$$\frac{dy}{dx} = -1$$

$$r_y^2 x = r_x^2 y$$



- Step in *x* while

$$r_y^2 x < r_x^2 y$$

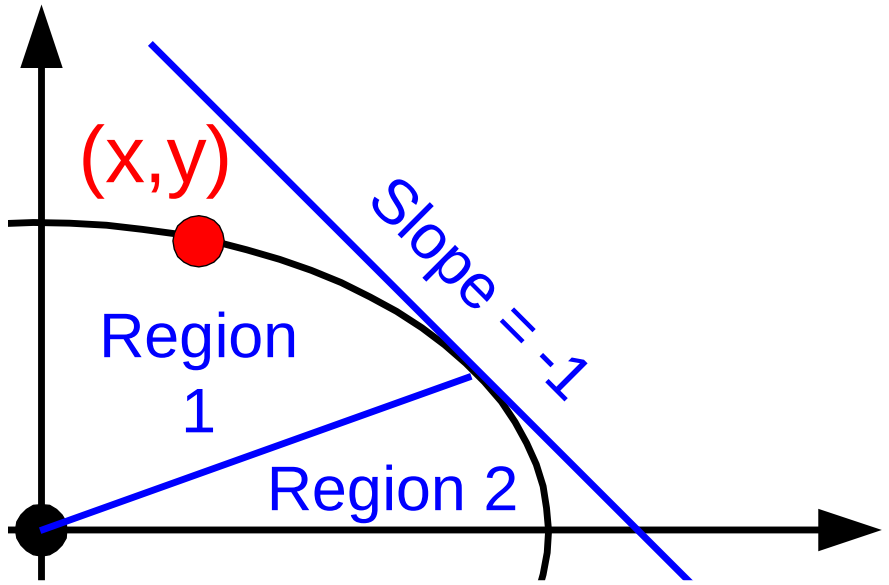
- Switch to steps in *y* when

$$r_y^2 x \geq r_x^2 y$$

Midpoint Algorithm (initializing)

- Similar to circles
- The initial value for region 1

$$D_{init1} = f(1, r_y - \tfrac{1}{2})$$
$$= r_y^2 - r_x^2 r_y + \tfrac{1}{4} r_x^2$$



- The initial value for region 2

$$D_{init2} = f(x_p + \tfrac{1}{2}, y_p - 1)$$
$$= r_y^2 (x_p + \tfrac{1}{2})^2 + r_x^2 (y_p - 1)^2 - r_x^2 r_y^2$$

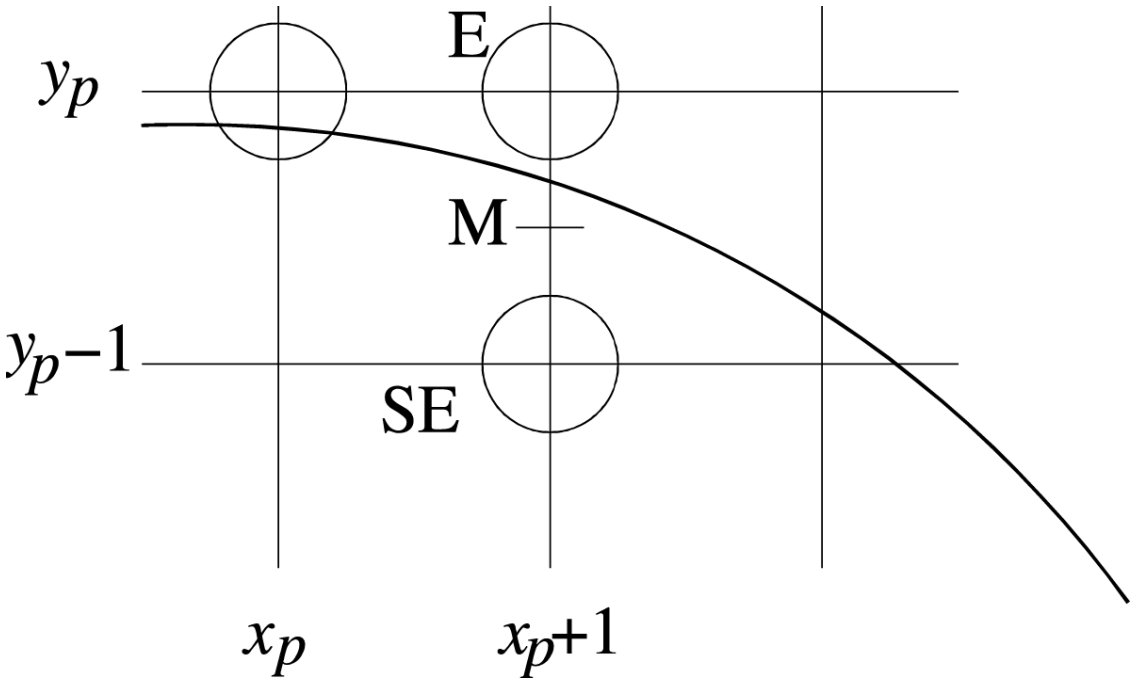
- We have initial values, now we need the increments

Making a Decision

- Computing the decision variable/ discriminant

$$D = f(x_p + 1, y_p - \tfrac{1}{2})$$
$$= r_y^2(x_p + 1)^2 + r_x^2(y_p - \tfrac{1}{2})^2 - r_x^2 r_y^2$$

- If $D < 0$ then M is *below* the arc, hence the E pixel is closer to the line.
- If $D \geq 0$ then M is *above* the arc, hence the SE pixel is closer to the line.



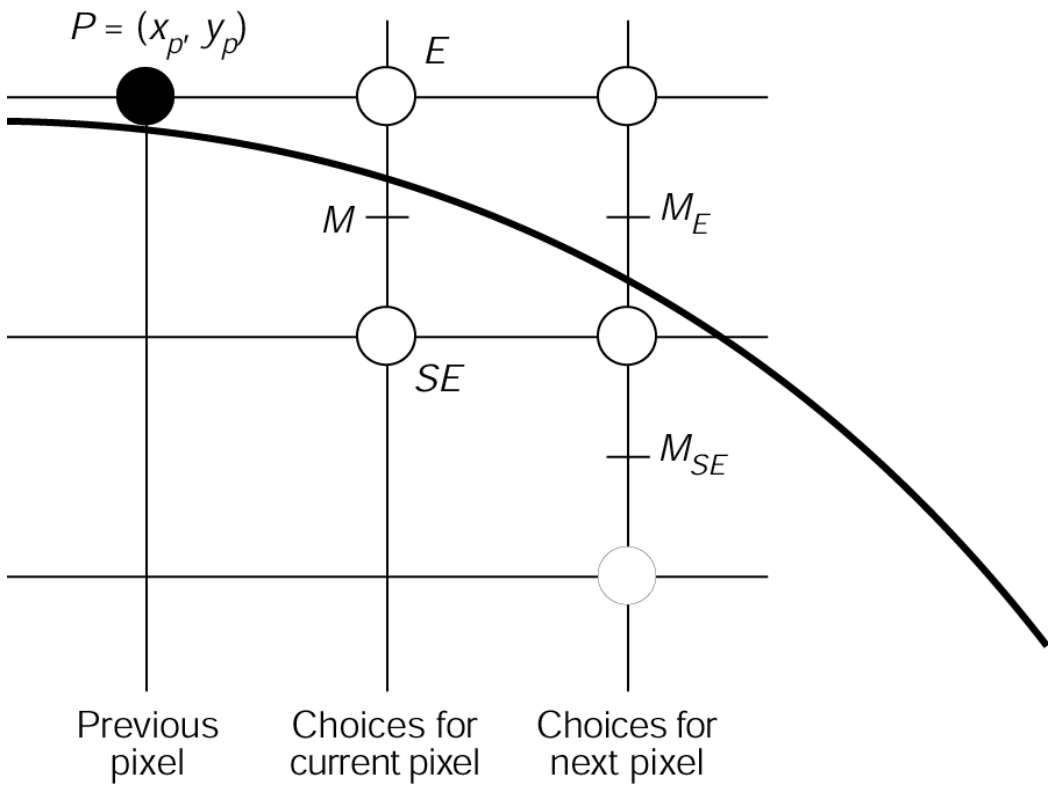
Computing the rate of change of Discriminant

E case

$$D_{new} = f(x_p + 2, y_p - \tfrac{1}{2})$$
$$= r_y^2(x_p + 2)^2 + r_x^2(y_p - \tfrac{1}{2})^2 - r_x^2 r_y^2$$
$$= D_{old} + r_y^2(2x_p + 3)$$

SE case

$$D_{new} = f(x_p + 2, y_p - \tfrac{3}{2})$$
$$= r_y^2(x_p + 2)^2 + r_x^2(y_p - \tfrac{3}{2})^2 - r_x^2 r_y^2$$
$$= D_{old} + r_y^2(2x_p + 3) + r_x^2(-2y_p + 2)$$



$$increment = \begin{cases} r_y^2(2x_p + 3) & D_{old} < 0 \\ r_y^2(2x_p + 3) + r_x^2(-2y_p + 2) & D_{old} \geq 0 \end{cases}$$

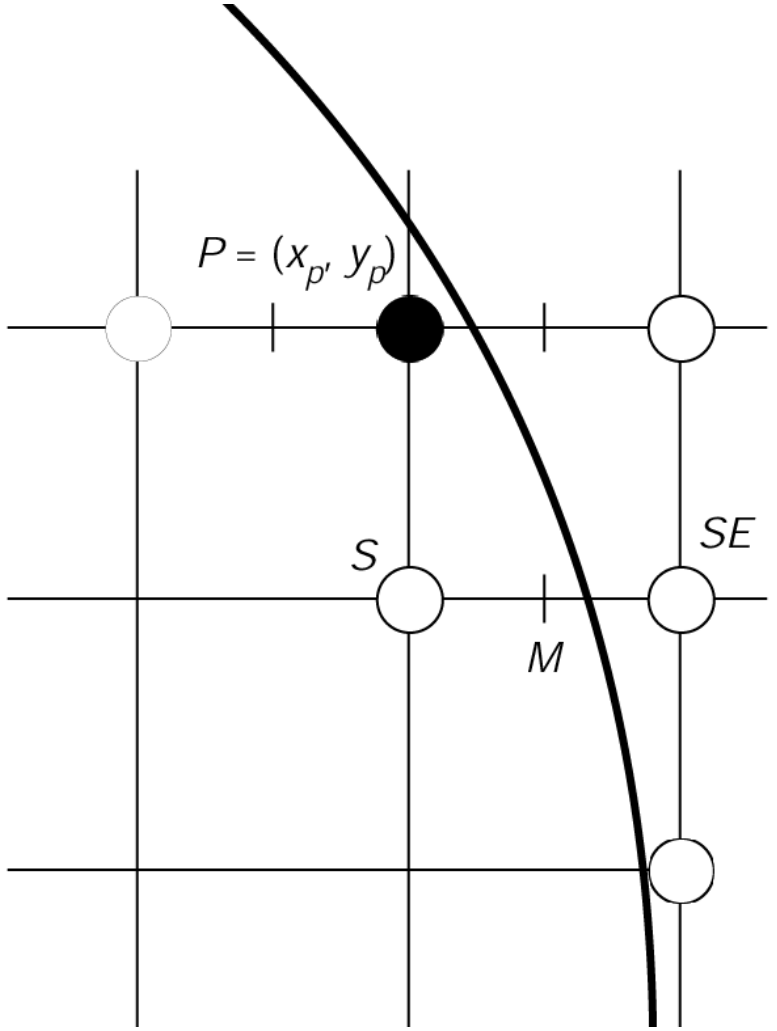
Computing the Increment in 2nd Region

- Decision variable in 2nd region

$$D = f(x_p + \tfrac{1}{2}, y_p - 1)$$
$$= r_y^2(x_p + \tfrac{1}{2})^2 + r_x^2(y_p - 1)^2 - r_x^2 r_y^2$$

- If $D < 0$ then M is *left of* the arc, hence the *SE* pixel is closer to the line.

- If $D \geq 0$ then M is *right of* the arc, hence the *S* pixel is closer to the line.



Computing the Increment in 2nd Region

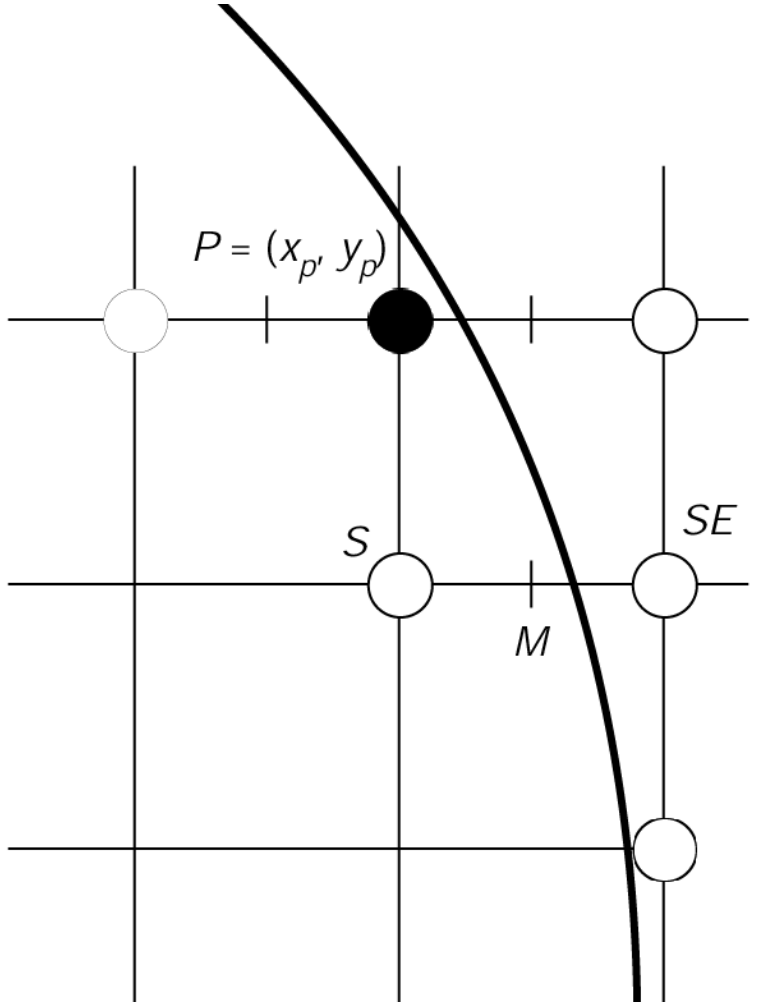
SE case

$$D_{new} = f(x_p + \tfrac{3}{2}, y_p - 2)$$
$$= r_y^2(x_p + \tfrac{3}{2})^2 + r_x^2(y_p - 2)^2 - r_x^2 r_y^2$$
$$= D_{old} + r_x^2(-2y_p + 3) + r_y^2(2x_p + 2)$$

S case

$$D_{new} = f(x_p + \tfrac{1}{2}, y_p - 2)$$
$$= r_y^2(x_p + \tfrac{1}{2})^2 + r_x^2(y_p - 2)^2 - r_x^2 r_y^2$$
$$= D_{old} + r_x^2(-2y_p + 3)$$

$$increment = \begin{cases} r_x^2(-2y_p + 3) + r_y^2(2x_p + 2) & D_{old} < 0 \\ r_x^2(-2y_p + 3) & D_{old} \geq 0 \end{cases}$$



Midpoint Algorithm for Ellipses

Region 1

Set first point to $(0, r_y)$

Set the Decision variable to

$$D_{init1} = r_y^2 - r_x^2 r_y + \tfrac{1}{4} r_x^2$$

Loop *(x = x+1)*

If $D < 0$ then pick *E* and

$$D \mathrel{+}= r_y^2(2x_p + 3)$$

If $D \geq 0$ then pick *SE* and

$$D \mathrel{+}= 2r_y^2 x_{k+1} - 2r_x^2 y_{k+1} + 3r_y^2 + 2r_x^2$$

$y = y - 1;$

Use symmetry to complete the ellipse

Until $r_y^2(x_k + 1) > r_x^2(y_k - ½)$

Region 2

Set first point to the last computed

Set the Decision variable to from previous value

Or, $D_{init2} = r_y^2(x_p + \tfrac{1}{2})^2 + r_x^2(y_p - 1)^2 - r_x^2 r_y$

Loop *(y = y - 1)*

If $D < 0$ then pick *SE* and

$$D \mathrel{+}= r_y^2(2x_p + 2) + r_x^2(-2y_p + 3)$$

x = x + 1;

If $D \geq 0$ then pick *S* and

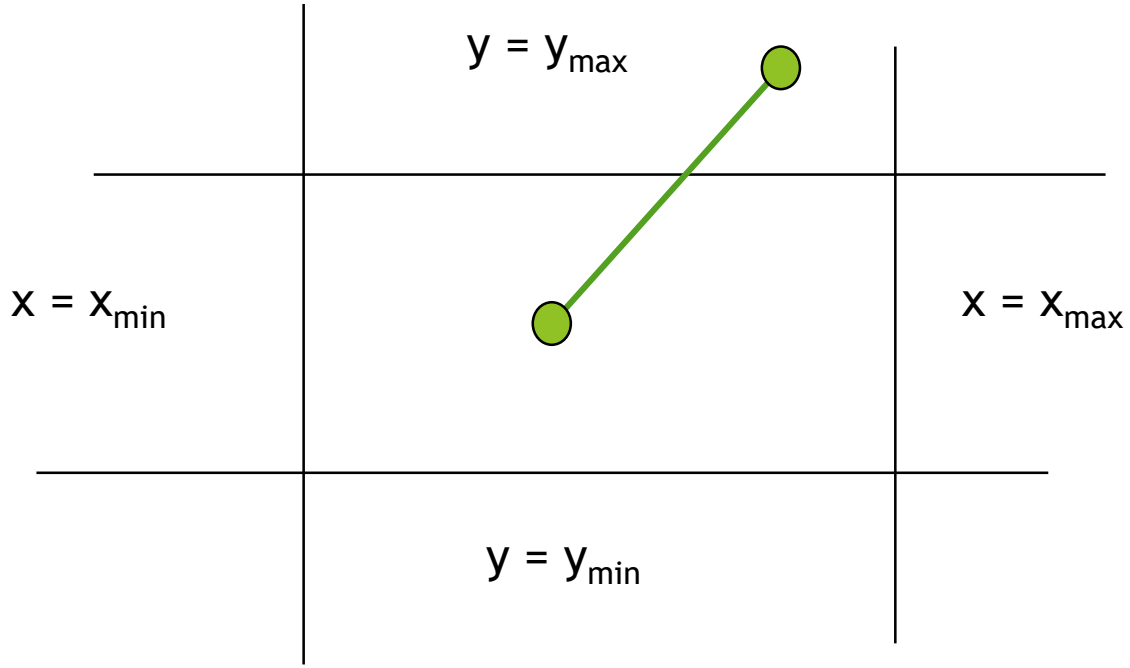
$$D \mathrel{+}= r_x^2(-2y_p + 3)$$

Use symmetry to complete the ellipse

Until $y < 0$

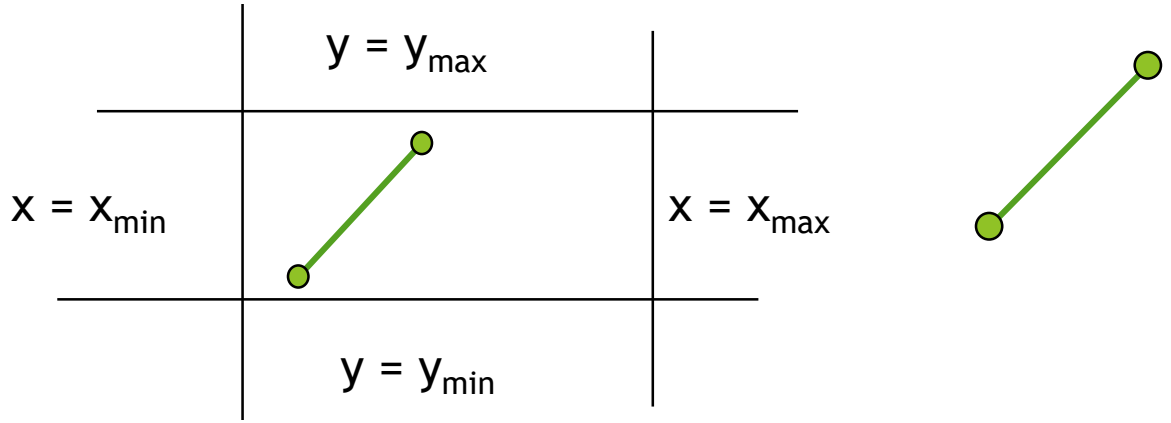
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



The Cases

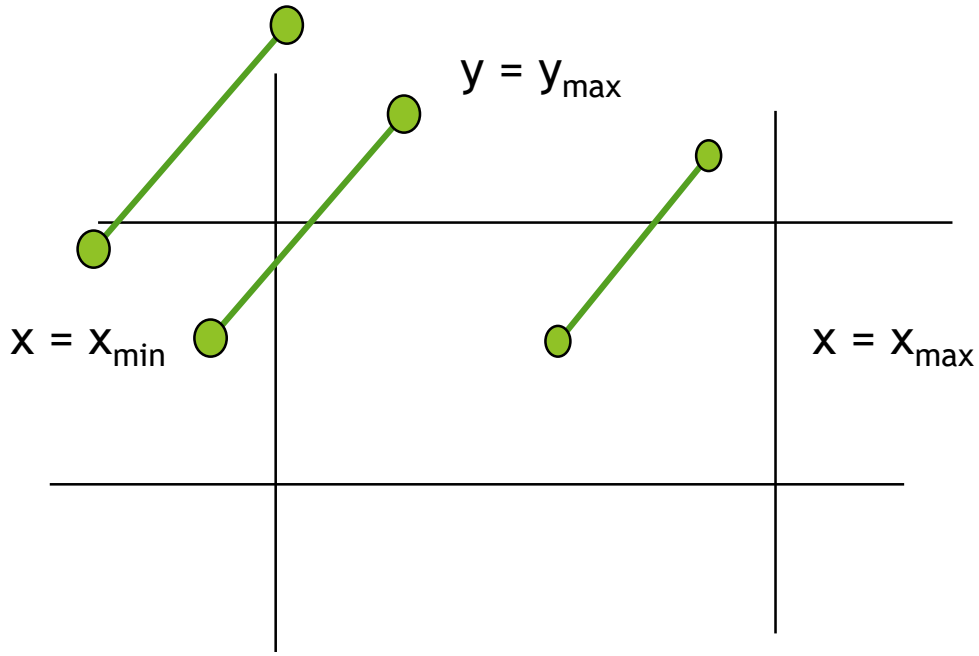
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

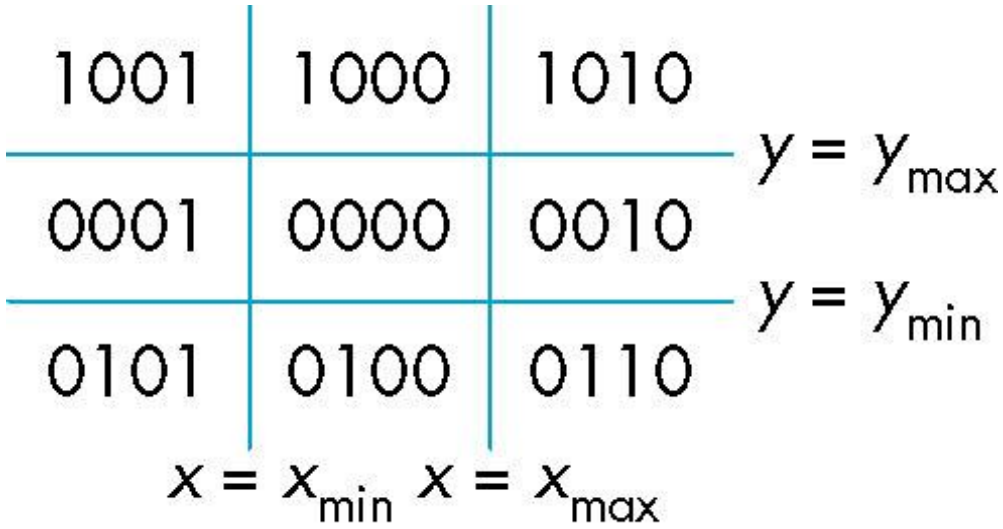
- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside
  - May have part inside
  - Must do at least one intersection



Defining Outcodes

- For each endpoint, define an outcode $b_0b_1b_2b_3$

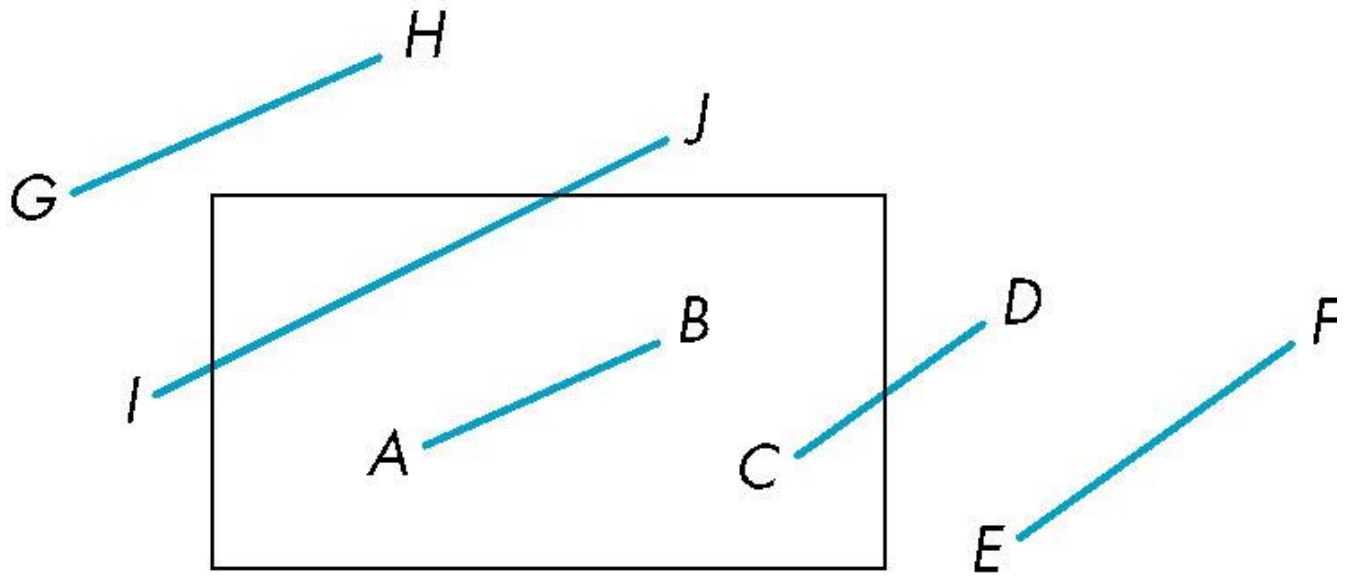
b_0 = 1 if $y > y_{max}$, 0 otherwise
b_1 = 1 if $y < y_{min}$, 0 otherwise
b_2 = 1 if $x > x_{max}$, 0 otherwise
b_3 = 1 if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions

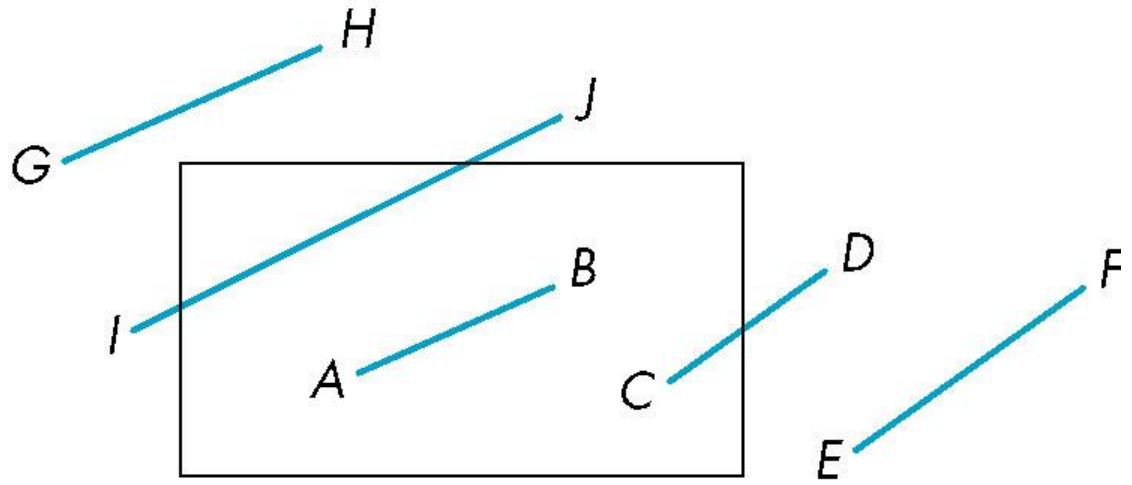
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - Accept line segment



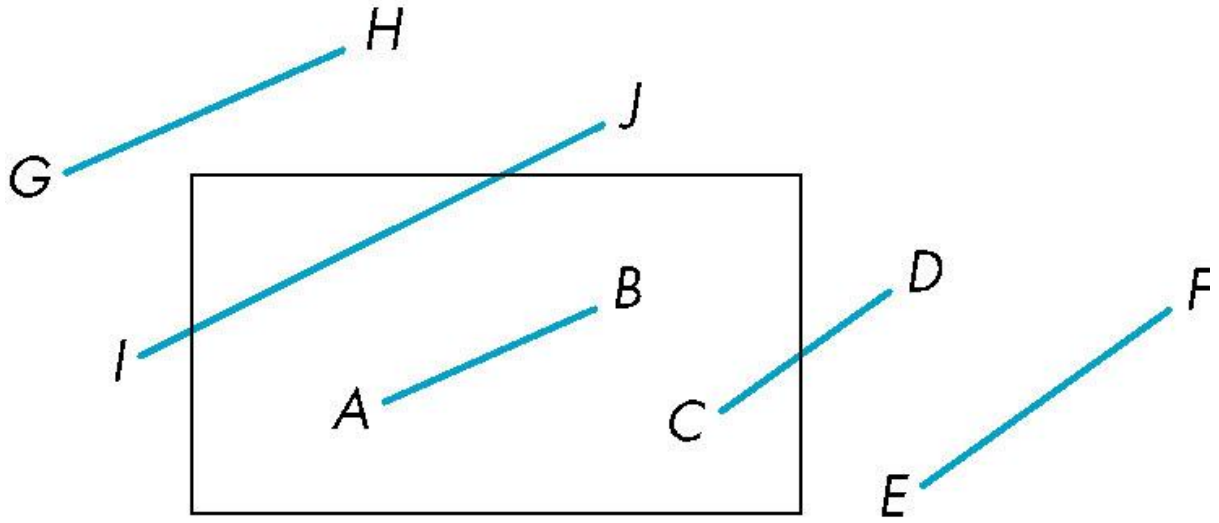
Using Outcodes

- CD: outcode (C) = 0, outcode(D) ≠ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two intersections



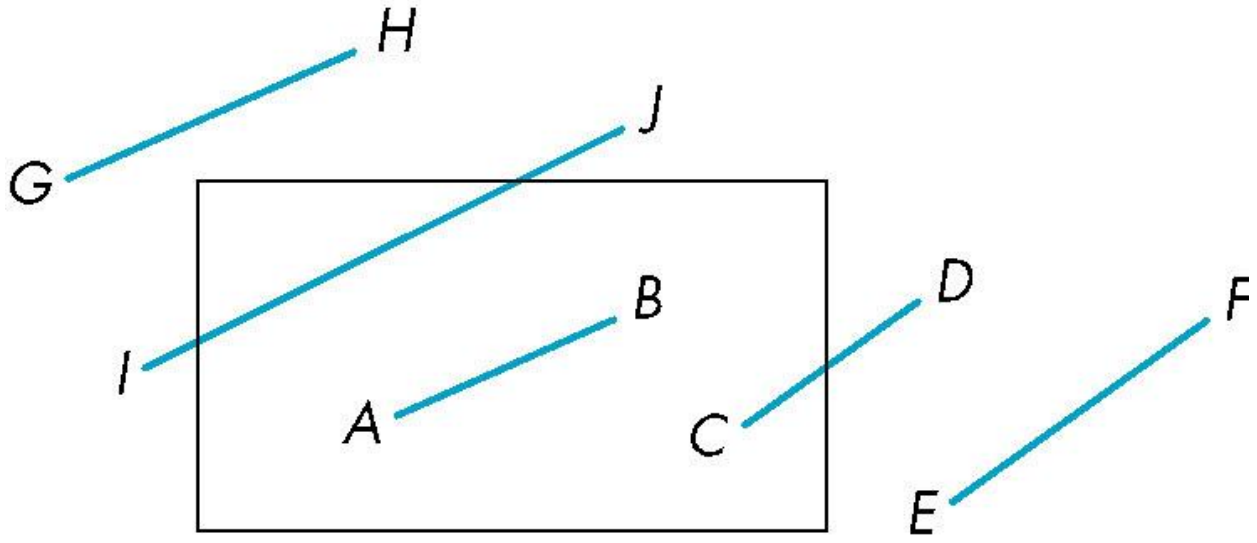
Using Outcodes

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) ≠ 0
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - reject



Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm

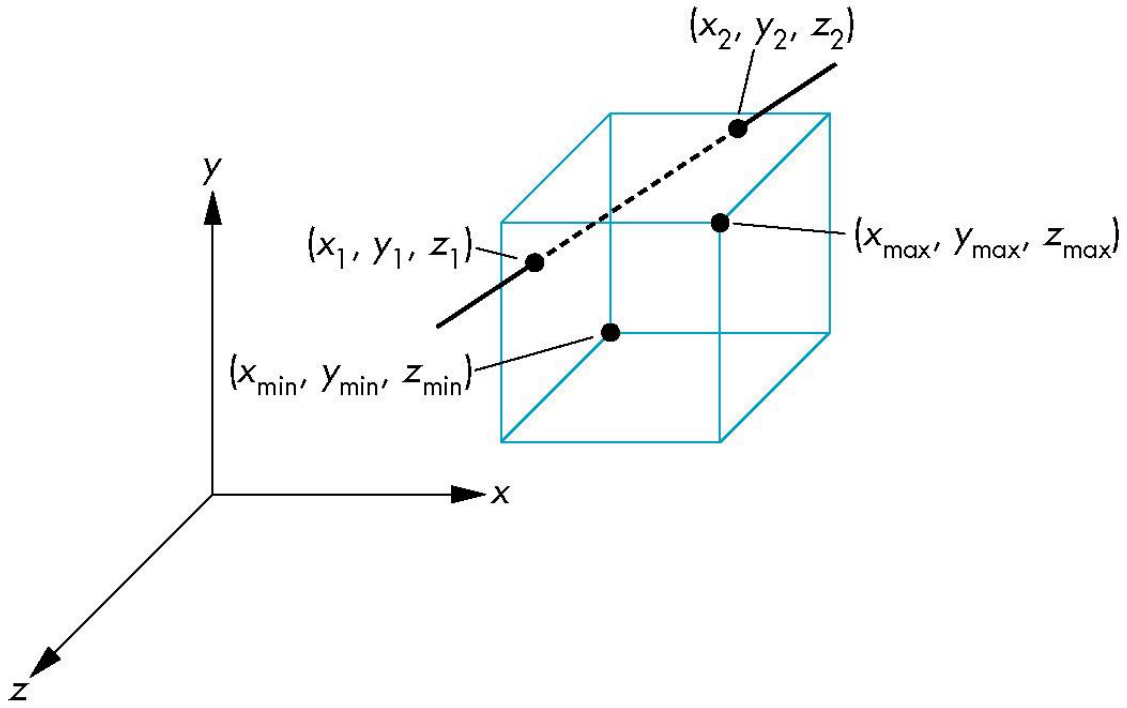


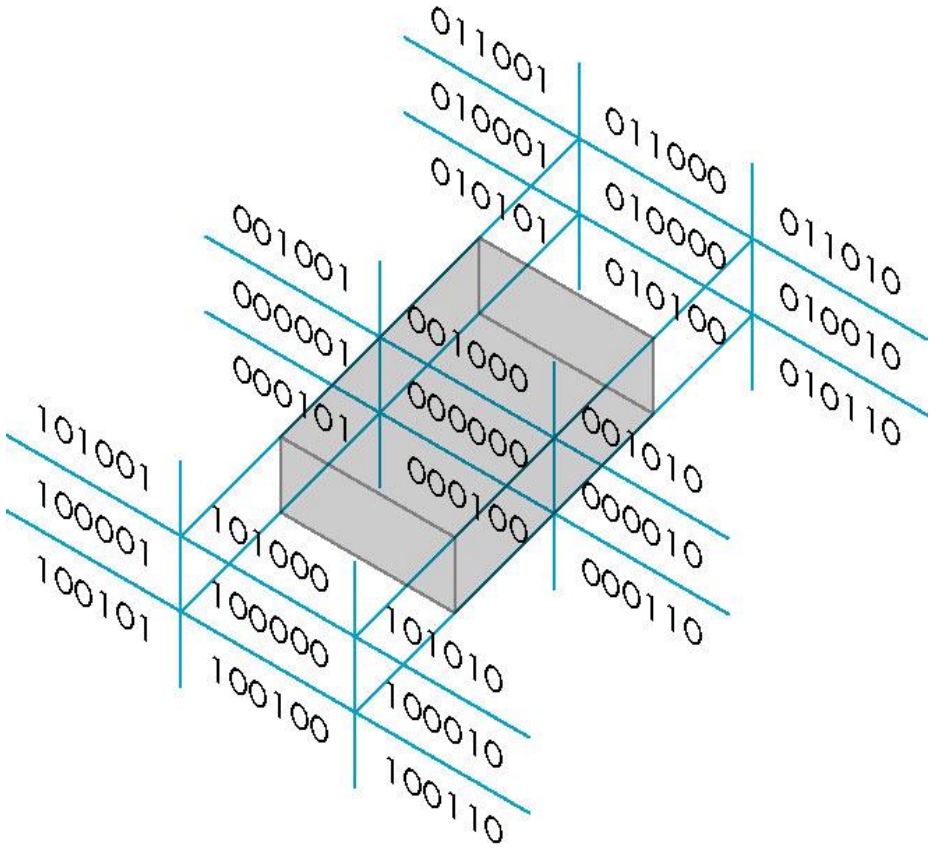
Efficiency

- In many applications, the clipping window is small relative to the size of the whole data base
  - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes

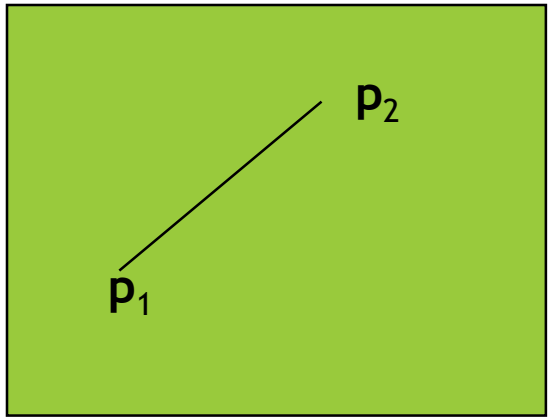




Liang-Barsky Clipping

$$p(t) = (1-t)p_1 + tp_2 \quad 1 \geq t \geq 0$$

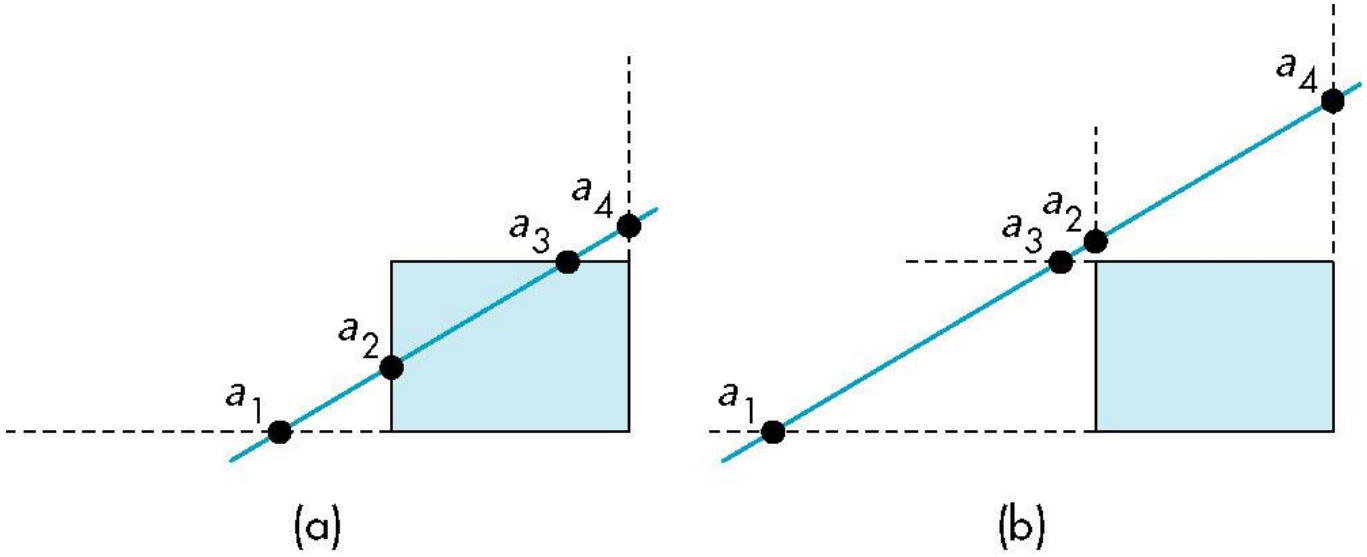
- Consider the parametric form of a line segment



- We can distinguish between the cases by looking at the ordering of the values of t where the line determined by the line segment crosses the lines that determine the window

Liang-Barsky Clipping

- In (a): $\alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$
  - Intersect right, top, left, bottom: shorten
- In (b): $\alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$
  - Intersect right, left, top, bottom: reject

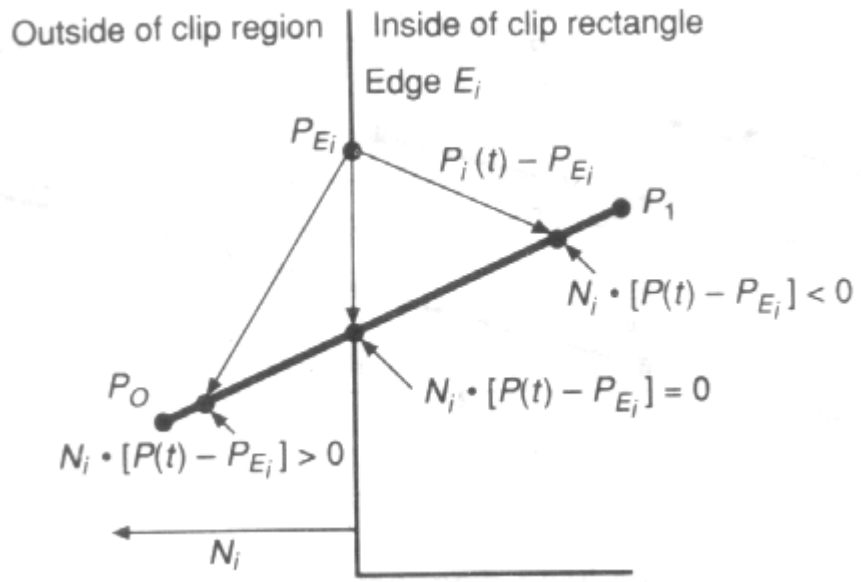


Advantages

- Can accept/reject as easily as with Cohen-Sutherland
- Using values of α, we do not have to use algorithm recursively as with C-S
- Extends to 3D

Parametric Line-Clipping Algorithm

- Introduced by Cyrud and Beck in 1978
- Efficiently improved by Liang and Barsky
- Essentially find the parameter t from $P(t) = P_0 + (P_1 - P_0)t$



$$N_i \cdot [P(t) - P_{Ei}] = 0$$

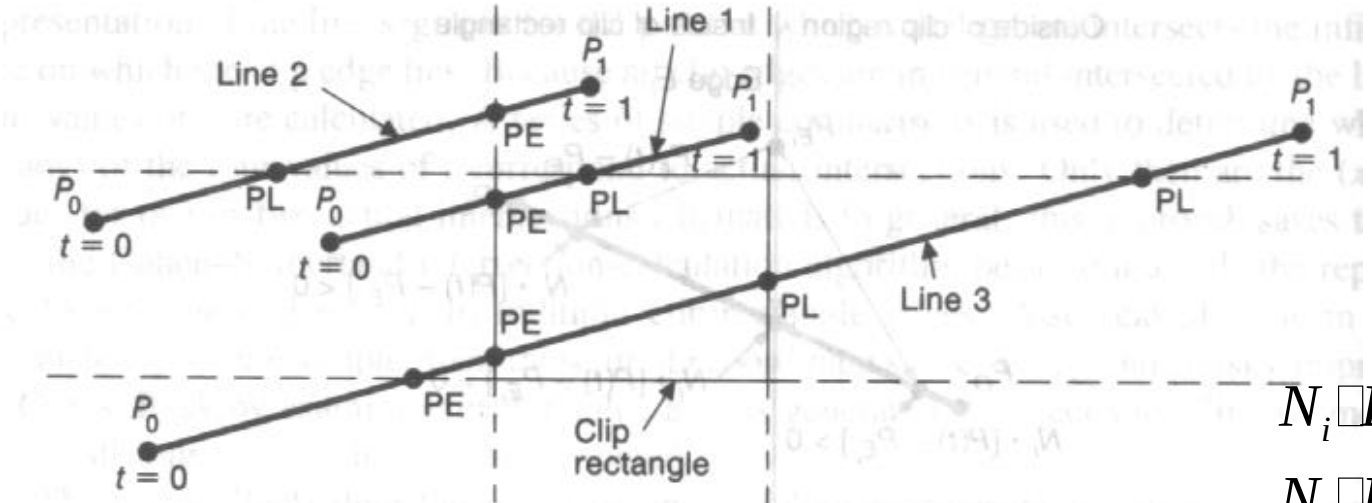
$$N_i \cdot [P_0 + (P_1 - P_0)t - P_{Ei}] = 0$$

$$N_i \cdot [P_0 - P_{Ei}] + N_i \cdot [P_1 - P_0]t = 0$$

$$t = -\frac{N_i \cdot [P_0 - P_{Ei}]}{N_i \cdot D}$$

$$where\ D = (P_1 - P_0)$$

Parametric Line-Clipping Algorithm (cont.)



$$N_i \cdot D < 0 \Rightarrow PE(angle > 90)$$

$$N_i \cdot D > 0 \Rightarrow PL(angle < 90)$$

- Formally, intersections can be classified as PE (potentially entering) and PL (potentially leaving) on the basis of the angle between P_0P_1 and N_i
- Determine t_E or t_L for each intersection
- Select the line segment that has maximum t_E and minimum t_L
- If $t_E > t_L$, then trivially rejected

Parametric Line-Clipping Algorithm (cont.)

TABLE 3.1 CALCULATIONS FOR PARAMETRIC LINE CLIPPING ALGORITHM*

Clip edge$_i$	Normal N_i	P_{E_i}	$P_0 - P_{E_i}$	$t = \frac{N_i \cdot (P_0 - P_{E_i})}{-N_i \cdot D}$
left: $x = x_{min}$	$(-1, 0)$	(x_{min}, y)	$(x_0 - x_{min}, y_0 - y)$	$\frac{-(x_0 - x_{min})}{(x_1 - x_0)}$
right: $x = x_{max}$	$(1, 0)$	(x_{max}, y)	$(x_0 - x_{max}, y_0 - y)$	$\frac{(x_0 - x_{max})}{-(x_1 - x_0)}$
bottom: $y = y_{min}$	$(0, -1)$	(x, y_{min})	$(x_0 - x, y_0 - y_{min})$	$\frac{-(y_0 - y_{min})}{(y_1 - y_0)}$
top: $y = y_{max}$	$(0, 1)$	(x, y_{max})	$(x_0 - x, y_0 - y_{max})$	$\frac{(y_0 - y_{max})}{-(y_1 - y_0)}$

*The

Cyrus-Beck Algorithm (Pseudocode)

```
precalculate N_i and select a P_Ei for each edge;

for (each line segment to be clipped) {
    if (P1 == P0)
        line is degenerate so clip as a point;
    else {
        t_E = 0; t_L = 1;
        for (each candidate intersection with a clip edge) {
            if (N_i • D != 0) {    /* Ignore edges parallel to line for now */
                calculate t;
                use sign of N_i • D to categorize as PE or PL;
                if (PE) t_E = max (t_E, t);
                if (PL) t_L = min (t_L, t);
            }
        }
        if (t_E > t_L)
            return NULL;
        else
            return P(t_E) and P(t_L) as true clip intersections;
    }
}
```

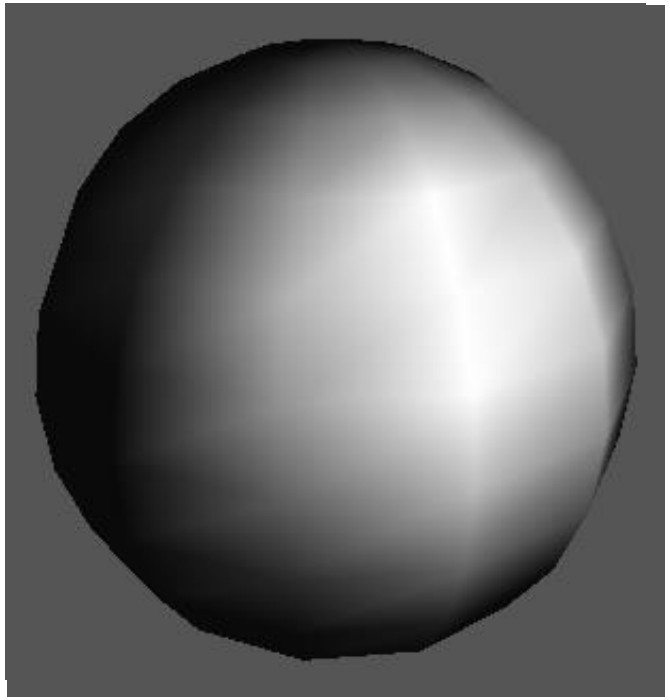
Lighting and Shading

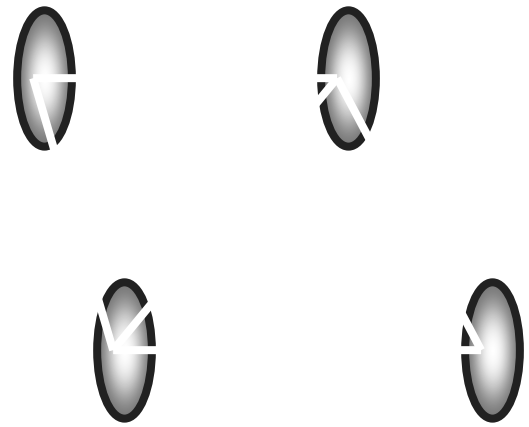
Outline

- Lighting
- Lighting models (Local and Global)
  - Ambient
  - Diffuse
  - Specular
- Surface Rendering Methods

What we know

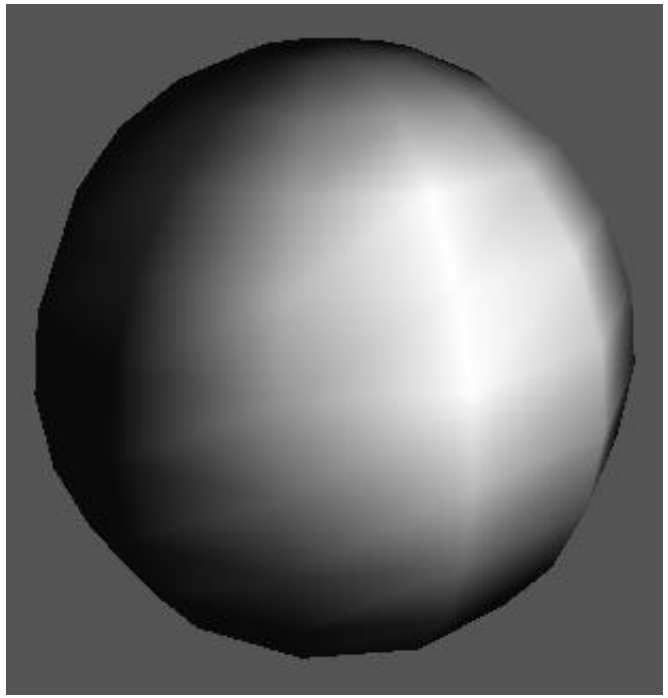
- We already know how to render the world from a viewpoint.





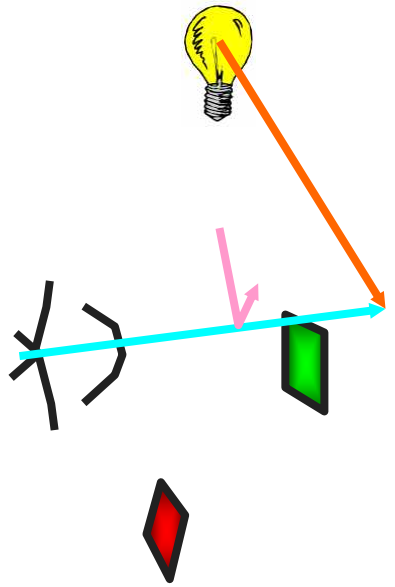
“Lighting”

- Two components:
  - **Lighting Model** or **Shading Model** - how we calculate the intensity at a point on the surface
  - **Surface Rendering Method** - How we calculate the intensity at each pixel



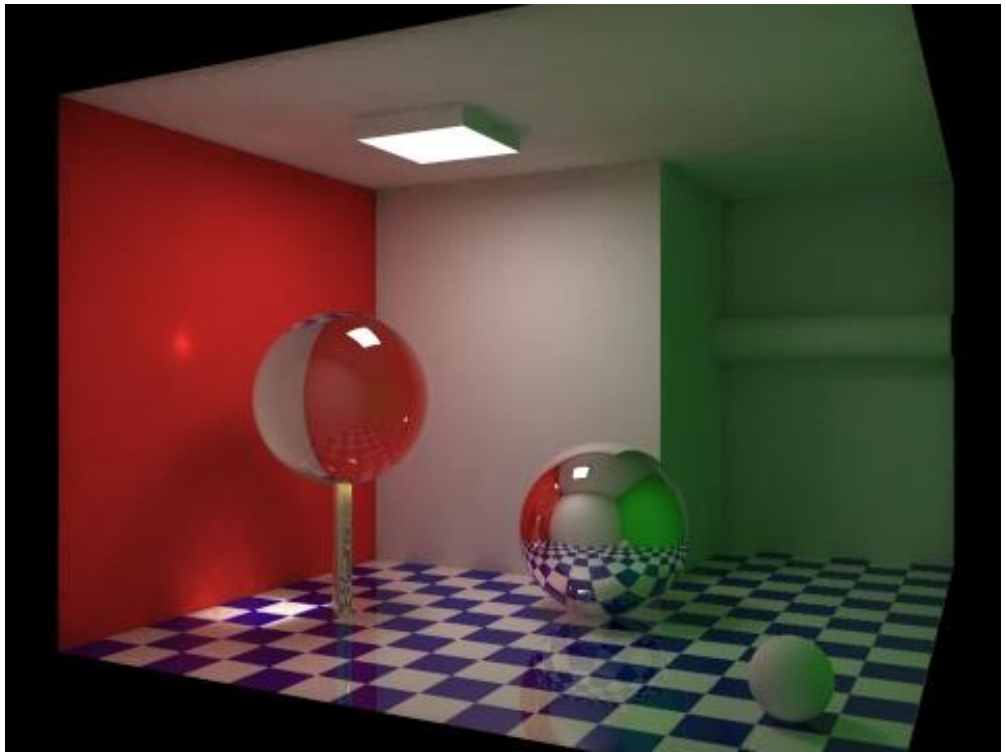
Garbage

- Illumination - the transport of light from a source to a point via direct and indirect paths
- Lighting - computing the luminous intensity for a specified 3D point,
- Shading - assigning colors to pixels based on lighting
- Illumination Models:
  - Empirical - approximations to observed light properties (Local Model)
  - Physically based - applying physics properties of light and its interactions with matter (Global Model)



The lighting problem…

- What are we trying to solve?
- **Global illumination** – the transport of light within a scene.
- What factors play a part in how an object is "lit"?
- Let's examine different items here…

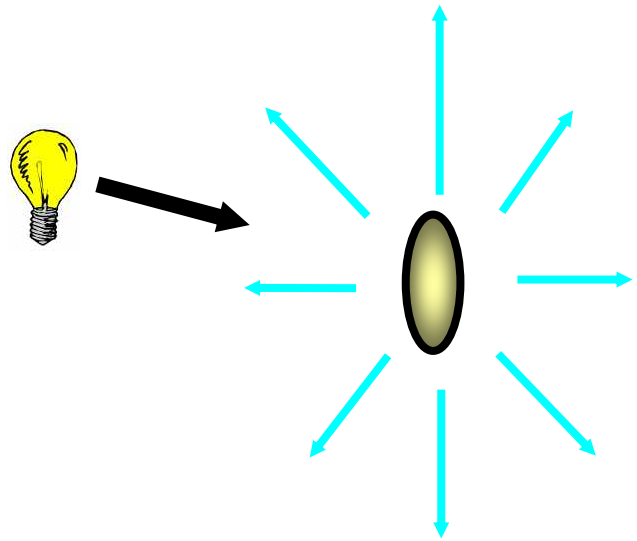


Two components

- Light Source Properties
  - Color (Wavelength(s) of light)
  - Shape
  - Direction
- Object Properties
  - Material
  - Geometry
  - Absorption

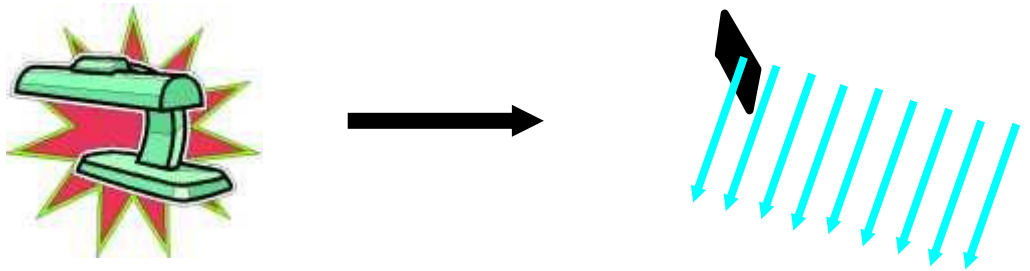
Light Source Properties

- Color
  - We usually *assume* the light has one wavelength
- Shape
  - point light source - approximate the light source as a 3D point in space. Light rays emanate in all directions.
    - good for small light sources (compared to the scene)
    - far away light sources



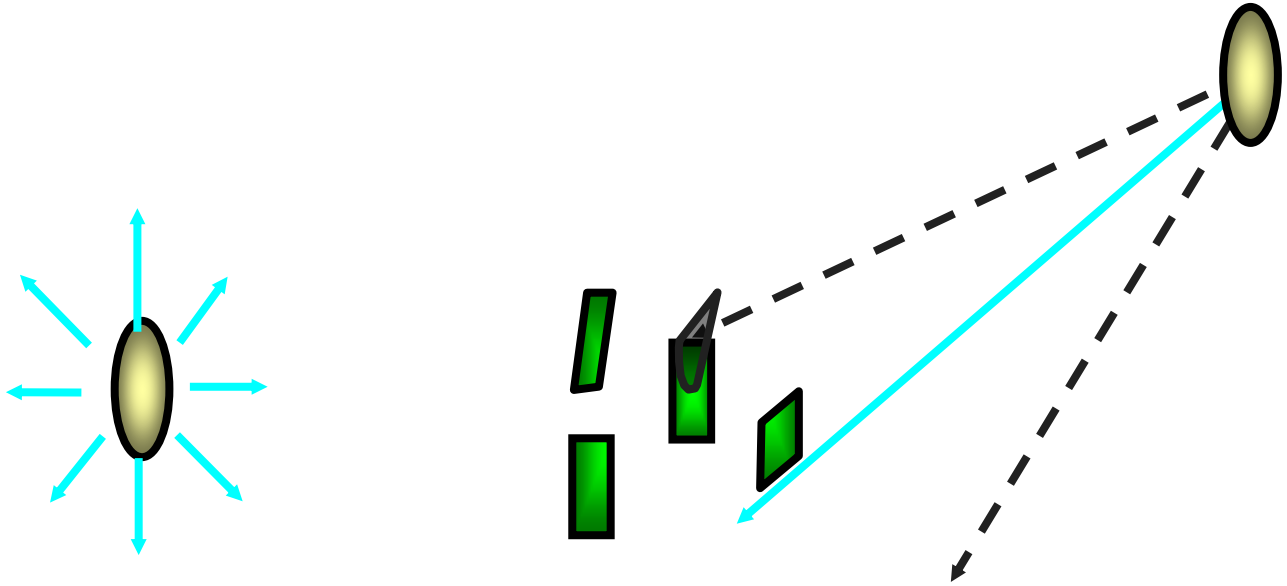
Distributed Lights

- Light Source Shape continued
  - distributed light source - approximating the light source as a 3D object. Light rays *usually* emanate in specific directions
    - good for larger light sources
    - area light sources



Light Source Direction

- In computer graphics, we usually treat lights as *rays* emanating from a source. The *direction* of these rays can either be:
  - Omni-directional (point light source)
  - Directional (spotlights)



Light Position

- We can specify the position of a light one of two ways, with an x, y, and z coordinate.
  - What are some examples?
  - These lights are called ***positional lights***
- Q: Are there types of lights that we can simplify?

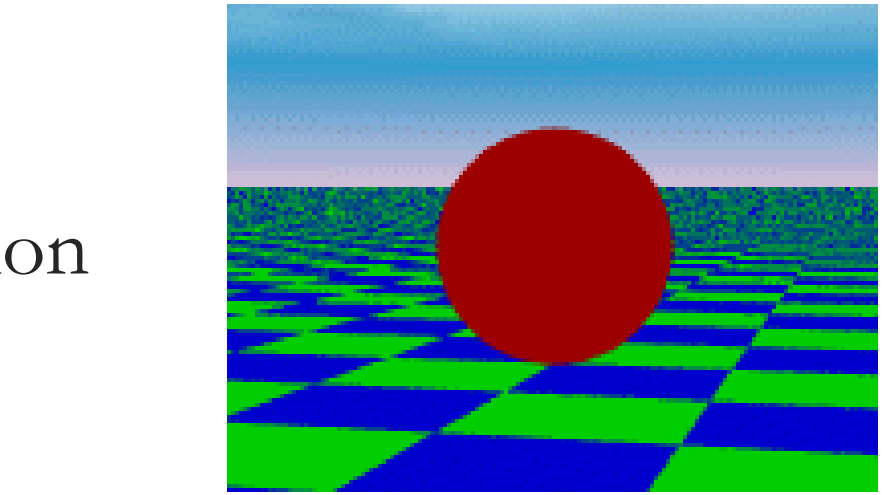
  A: Yep! Think about the sun. If a light is significantly far away, we can represent the light with *only* a direction vector. These are called ***directional lights.*** How does this help?

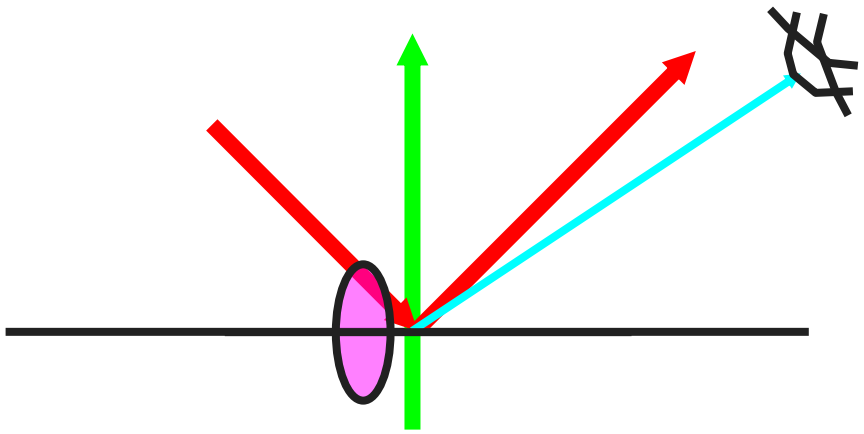
Contributions from lights

- We will breakdown what a light does to an object into three different components. This APPROXIMATES what a light does. To actually compute the rays is too expensive to do in real-time.
  - Light at a pixel from a light = Ambient + Diffuse + Specular contributions.
  - $I_{light} = I_{ambient} + I_{diffuse} + I_{specular}$

Ambient Term - Background Light

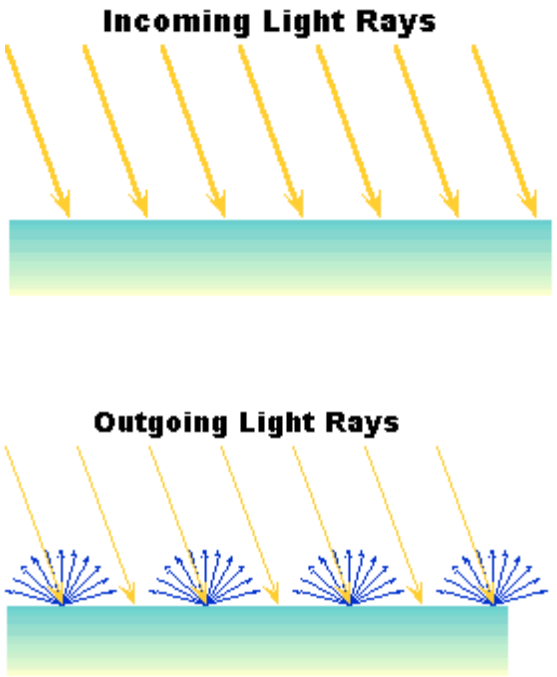
- The ambient term is a HACK!
- It represents the approximate contribution of the light to the general scene, regardless of location of light and object
- Indirect reflections that are too complex to completely and accurately compute
- $I_{ambient}$ = color





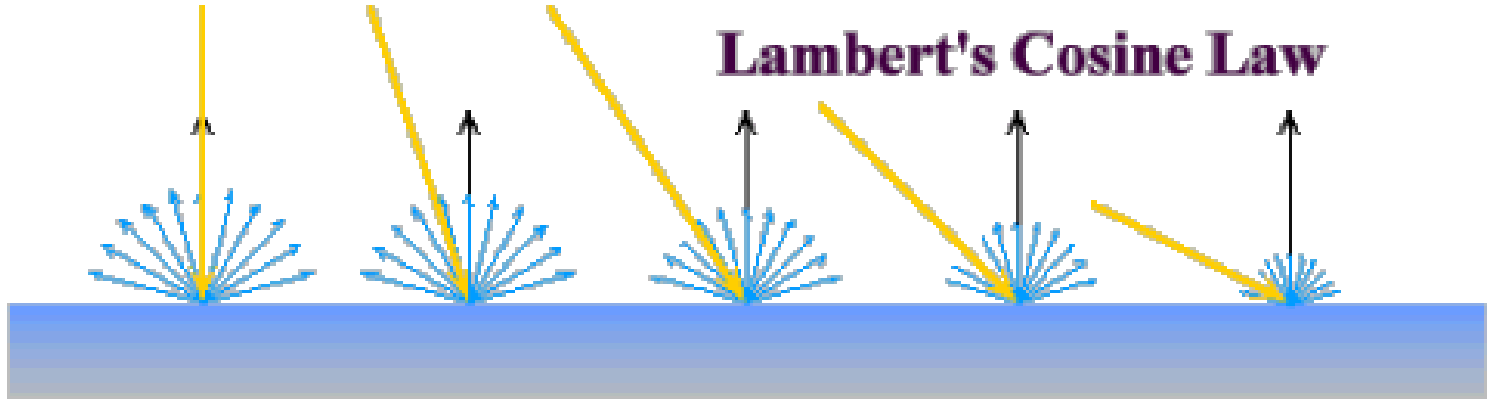
Diffuse Term

- Contribution that a light has on the surface, ***regardless of viewing direction***.
- Diffuse surfaces, on a microscopic level, are very rough. This means that a ray of light coming in has an equal chance of being reflected in *any* direction.
- What are some ideal diffuse surfaces?



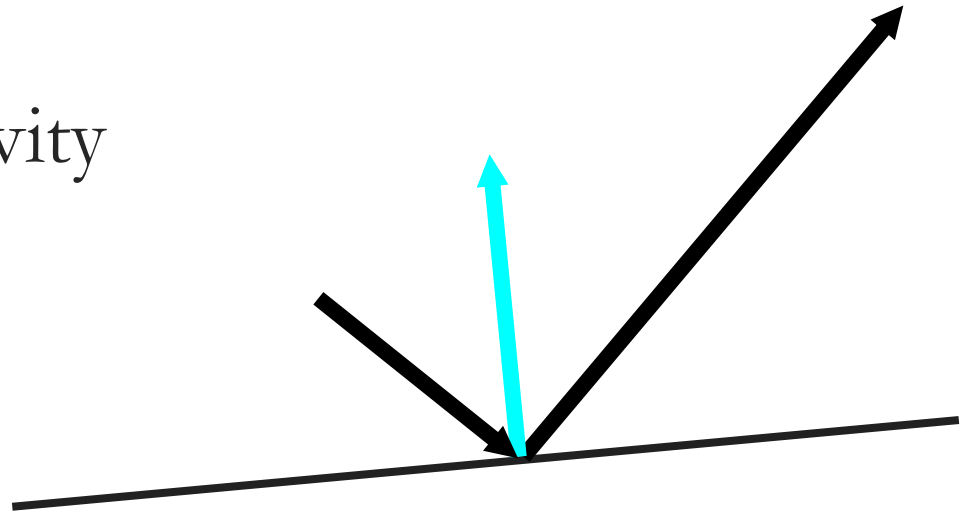
Lambert's Cosine Law

- Diffuse surfaces follow Lambert's Cosine Law
- Lambert's Cosine Law - reflected energy from a small surface area in a particular direction is proportional to the cosine of the angle between that direction and the surface normal.
- Think about surface area and # of rays



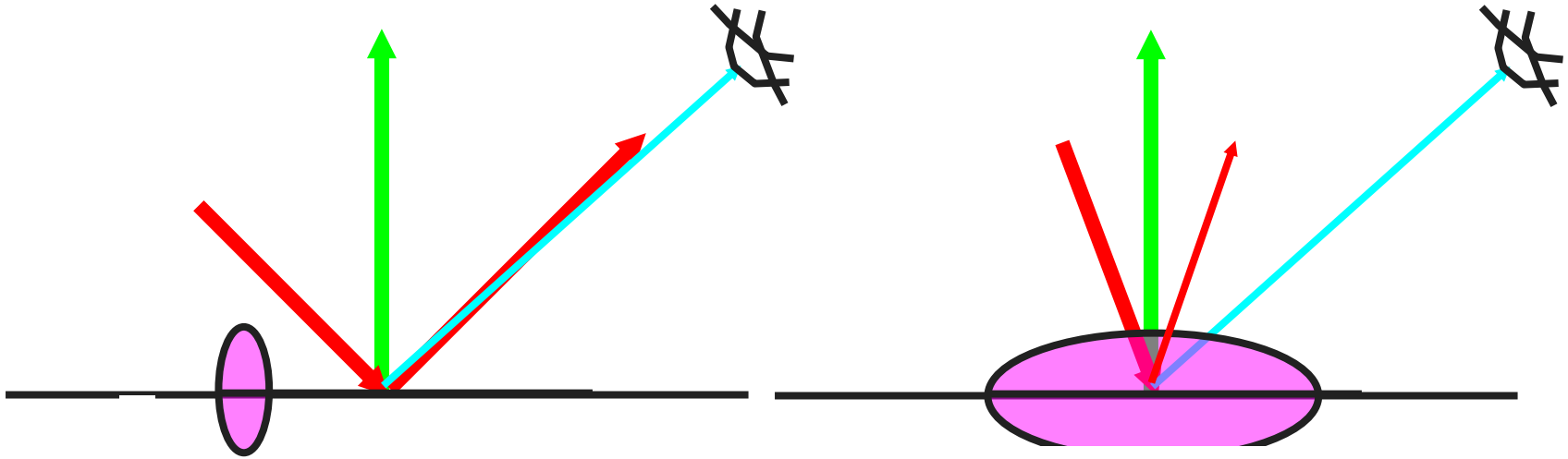
Diffuse Term

- To determine how much of a diffuse contribution a light supplies to the surface, we need the surface normal and the direction on the incoming ray
- What is the angle between these two vectors?
- $I_{diffuse} = k_d I_{light} \cos\theta = k_d I_{light}(N \cdot L)$
- I_{light} = diffuse (intensity) of light
- k_d [0..1] = surface diffuse reflectivity
- What CS are L and N in?
- How expensive is it?



Example

- What are the possible values for theta (and thus the dot product?)



Specular Reflection

- Specular contribution can be thought of as the "shiny highlight" of a plastic object.
- On a microscopic level, the surface is very smooth. Almost all light is reflected.
- What is an ideal purely specular reflector?
- What does this term depend on?

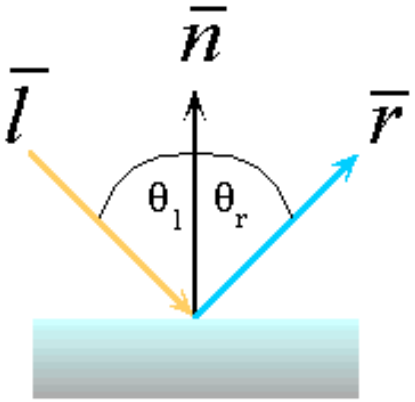
  Viewing Direction

  Normal of the Surface

Snell's Law

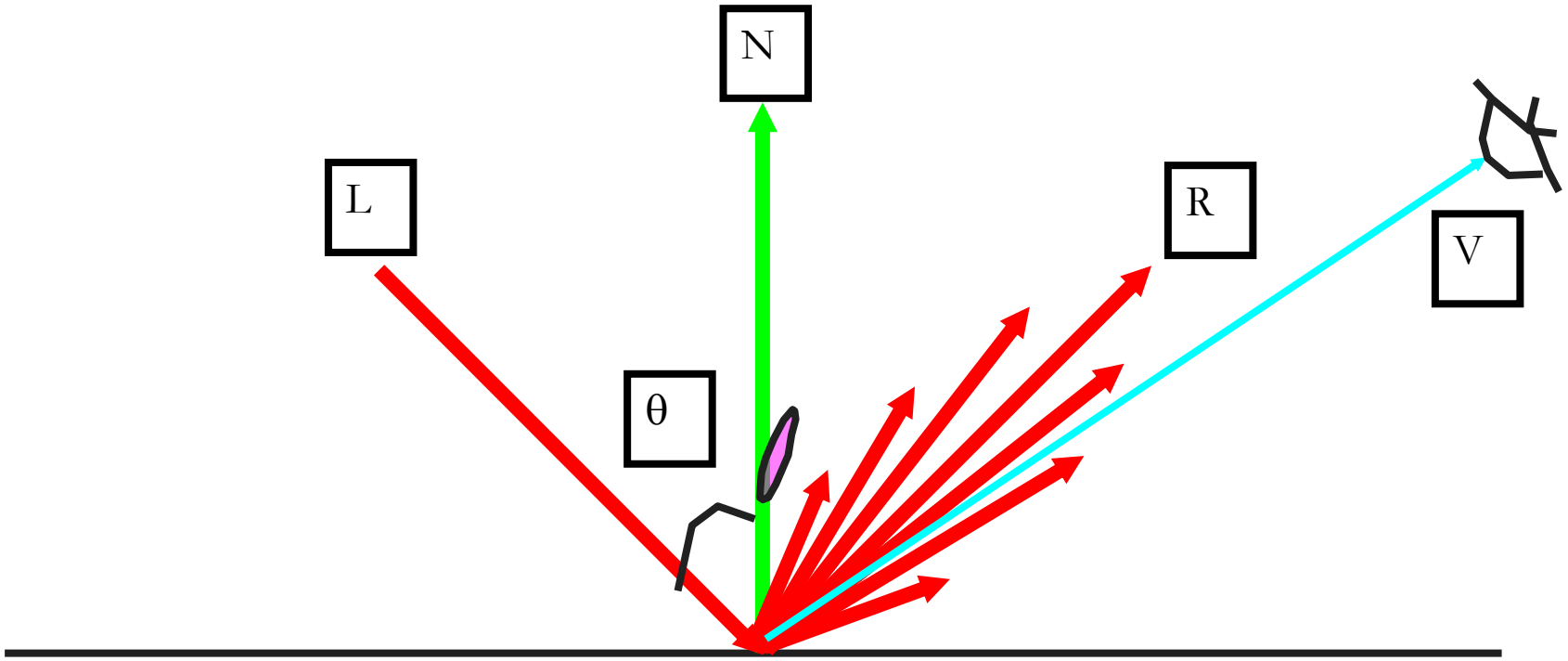
- Specular reflection applies Snell's Law.
  - The incoming ray, the surface normal, and the reflected ray all lie in a common plane.
  - The angle that the reflected ray forms with the surface normal is determined by the angle that the incoming ray forms with the surface normal, and the relative speeds of light of the mediums in which the incident and reflected rays propagate according to:
  - We assume $\theta_l = \theta_r$

$$n_l \sin\theta_l = n_r \sin\theta_r$$

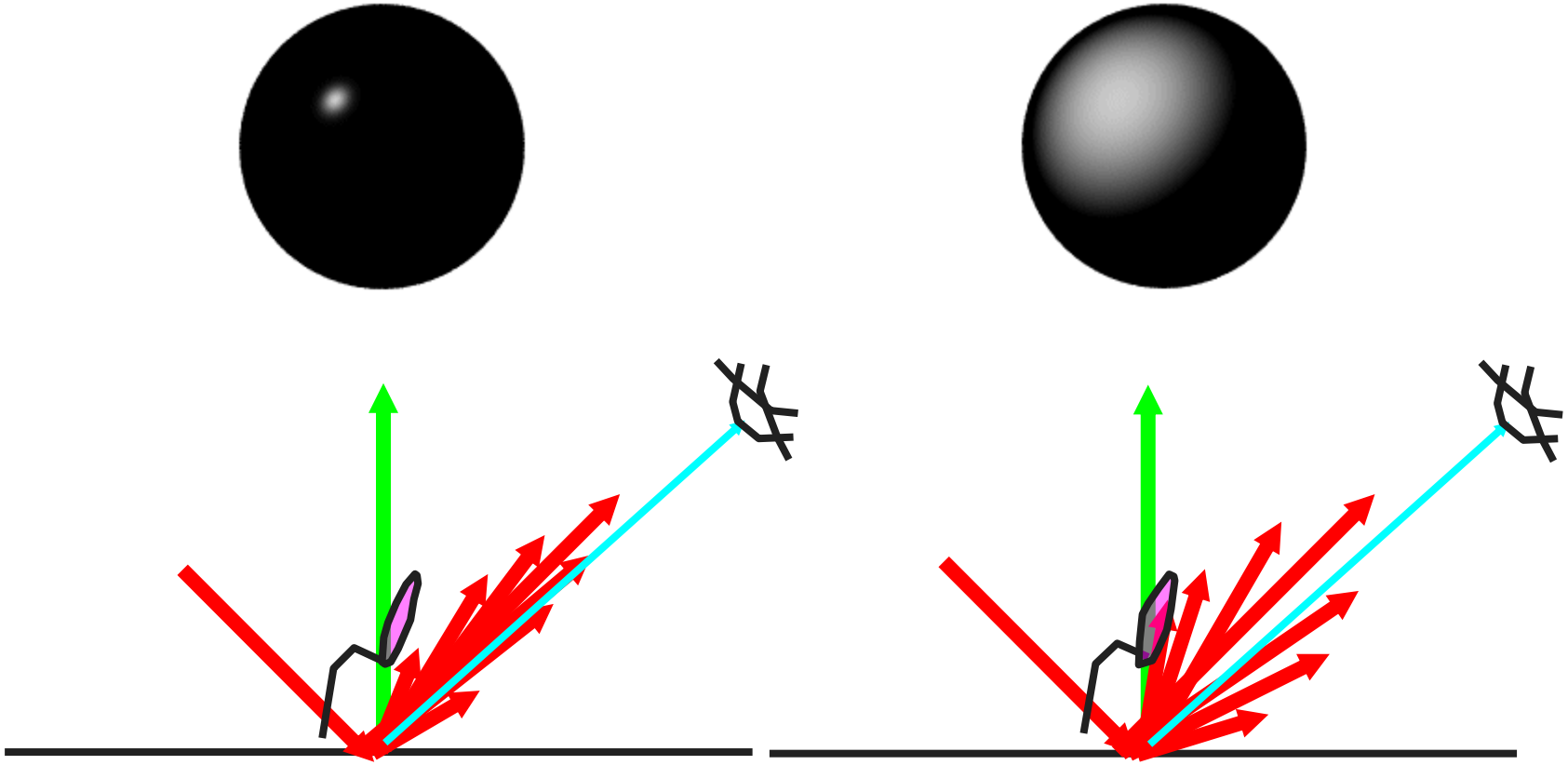


Snell's Law is for IDEAL surfaces

- Think about the amount of light reflected at different angles.

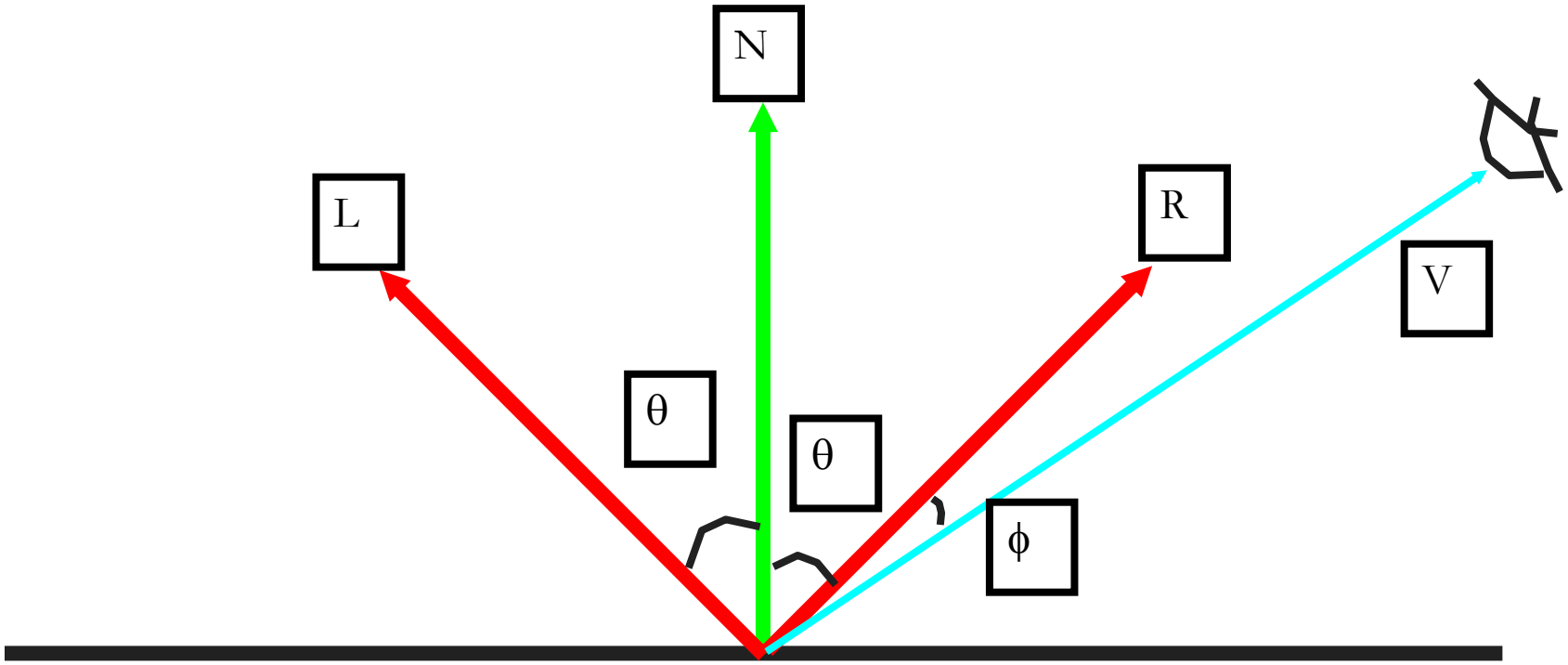


Different for shiny vs. dull objects



Snell's Law is for IDEAL surfaces

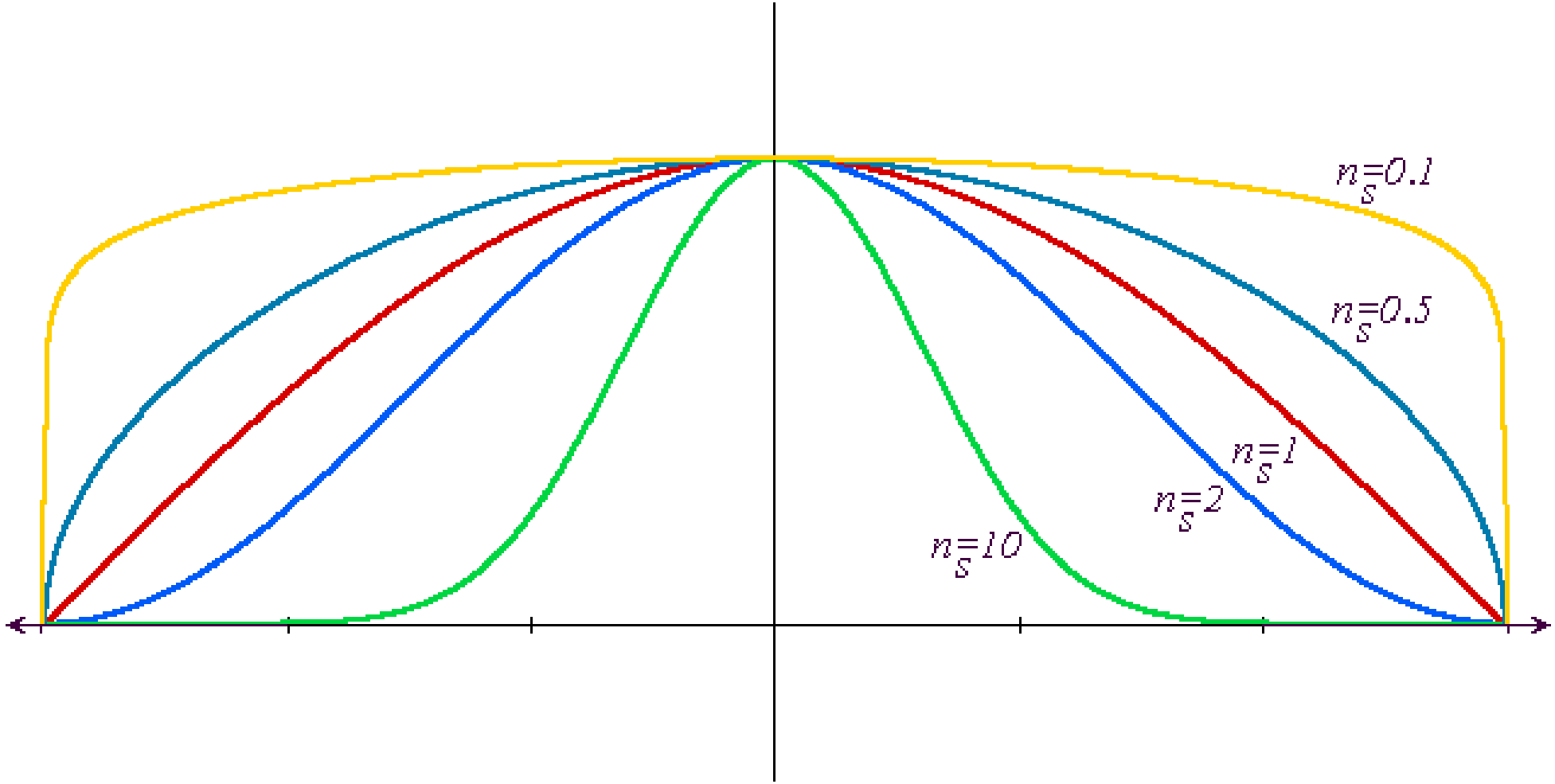
- Think about the amount of light reflected at different angles.



Phong Model/Phong Reflection Model

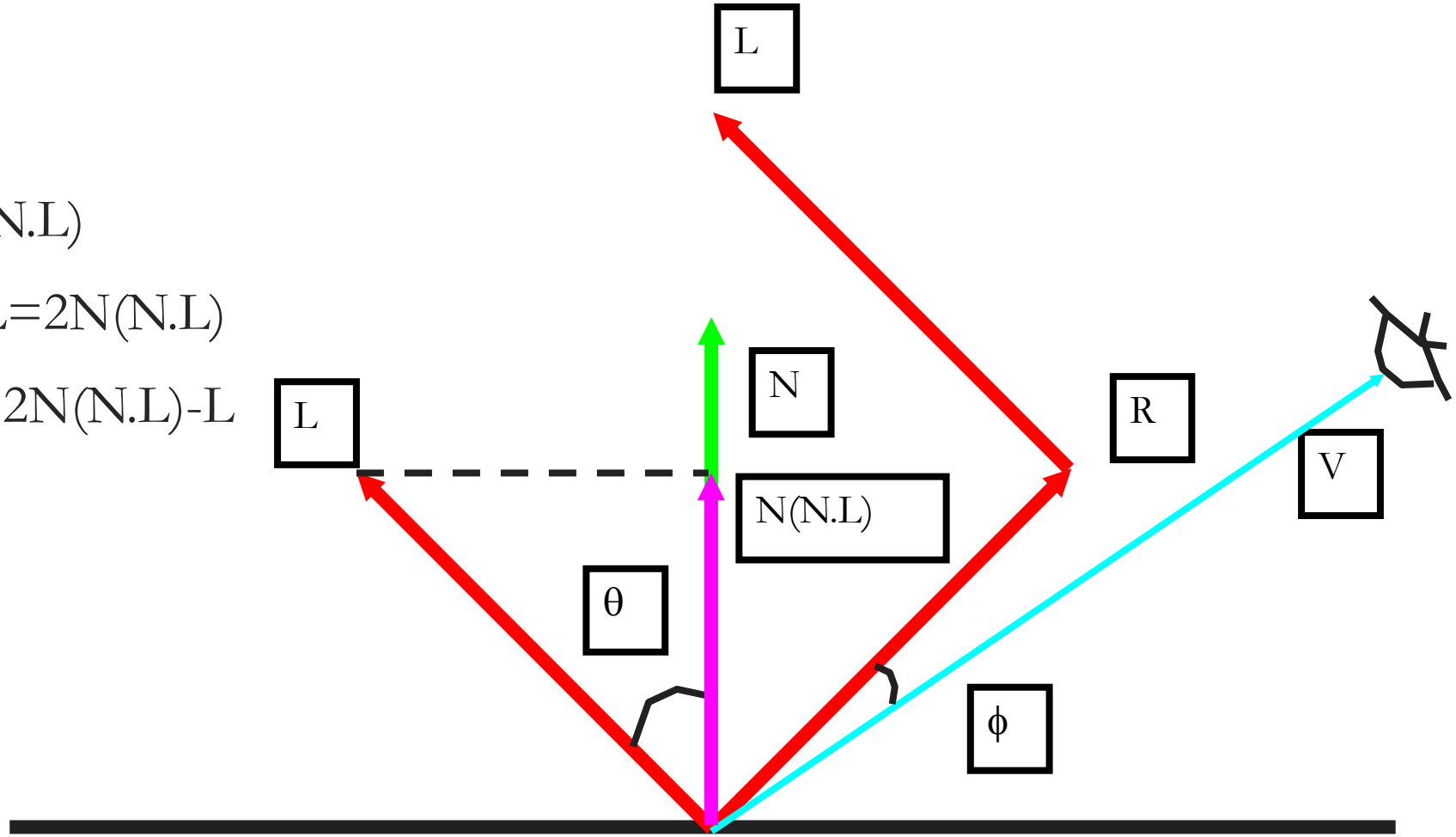
- An approximation is sets the intensity of specular reflection proportional to $(\cos \phi)^{\text{shininess}}$
- What are the possible values of $\cos \phi$?
- What does the value of *shininess* mean?
- How do we represent shinny or dull surfaces using the Phong model?
- What is the real thing we probably SHOULD do?
- $I_{specular} = k_s I_{light} (\cos \phi)^{\text{shininess}} = k_s I_{light} (V.R)^{\text{shininess}}$

Effect of the shininess value



How do we compute R?

- N*(N.L)
- R+L=2N(N.L)
- R = 2N(N.L)-L

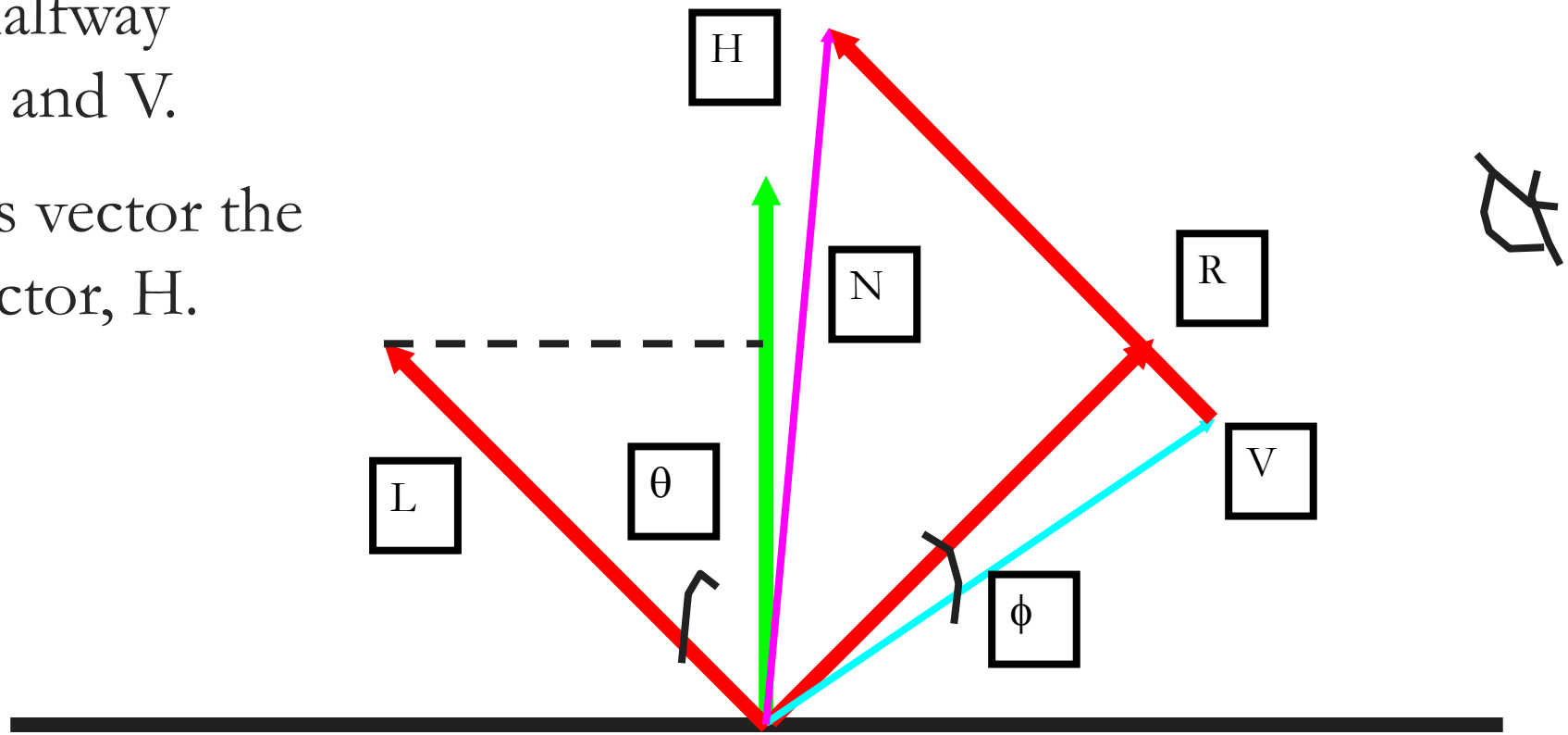


Simplify this

$$H = \frac{L + V}{|L + V|}$$

$$I_{specular} = k_s I_{light\_specularity} (N \cdot H)^{shininess}$$

- Instead of R, we compute halfway between L and V.
- We call this vector the halfway vector, H.



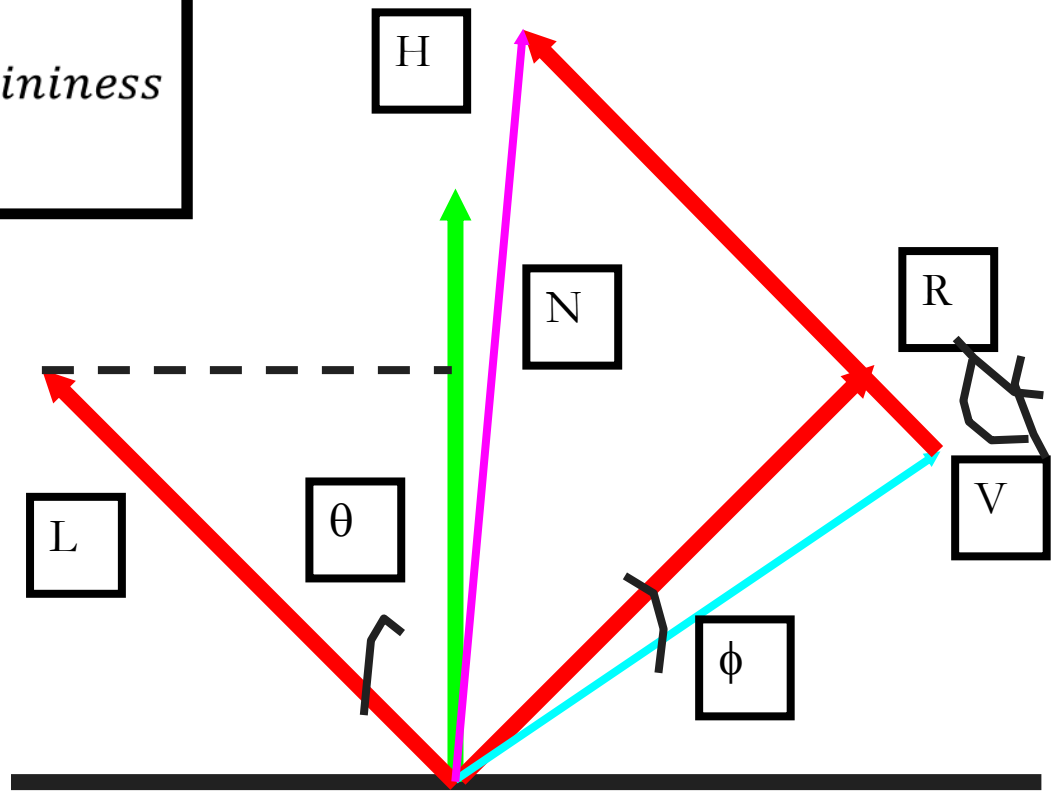
Let's compare the two

$$R = 2N(N \cdot L) - L$$
$$I_{specular} = k_s I_{light\_specularity}(V \cdot R)^{shininess}$$
$$H = \frac{L + V}{|L + V|}$$
$$I_{specular} = k_s I_{light\_specularity}(N \cdot H)^{shininess}$$

Q: Which vectors stay constant when viewpoint is far away?

A: V and L vectors -> H

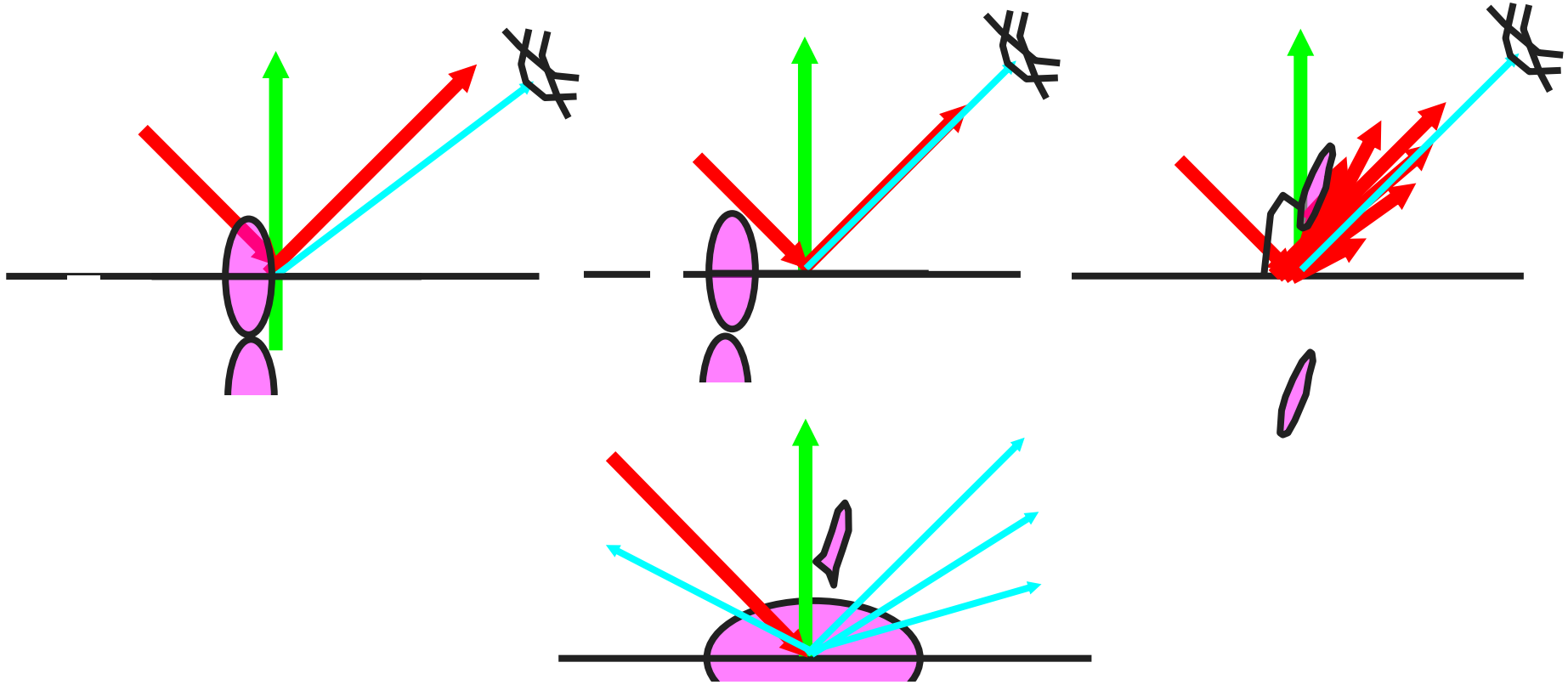
Q: What does this buy us?



Combining the terms

- Ambient - the combination of light reflections from various surfaces to produce a uniform illumination. Background light.
- Diffuse - uniform light scattering of light rays on a surface. Proportional to the "amount of light" that hits the surface. Depends on the surface normal and light vector.
- Sepecular - light that gets reflected. Depends on the light ray, the viewing angle, and the surface normal.

Ambient + Diffuse + Specular



Lighting Equation

$$I_{final} = I_{ambient} k_{ambient} + I_{diffuse} k_{diffuse} (N \cdot L) + I_{specular} k_{specular} (N \cdot H)^{shininess}$$

$$I_{final} = \sum_{l=0}^{lights-1} I_{l_{ambient}} k_{ambient} + I_{l_{diffuse}} k_{diffuse} (N \cdot L) + I_{l_{specular}} k_{specular} (N \cdot H)^{shininess}$$

$I_{lambient}$ = light source *l*'s ambient component

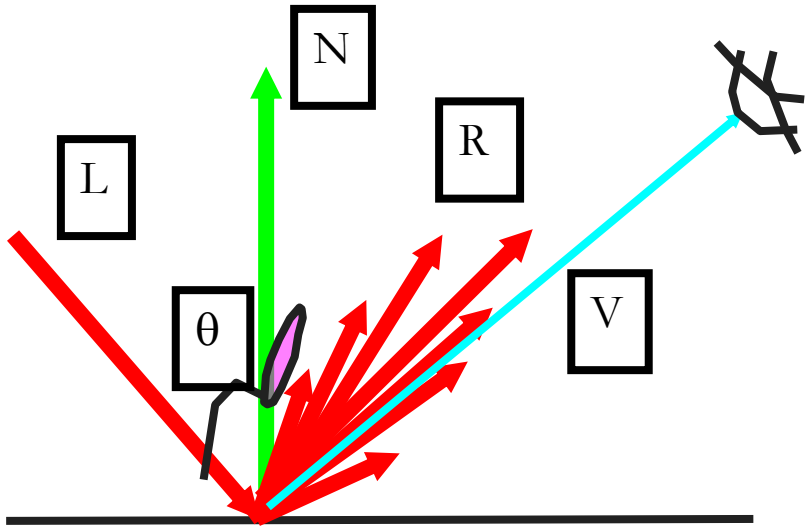
$I_{ldiffuse}$ = light source *l*'s diffuse component

$I_{lspecular}$ = light source *l*'s specular component

$k_{ambient}$ = surface material ambient reflectivity

$k_{diffuse}$ = surface material diffuse reflectivity

$k_{specular}$ = surface material specular reflectivity



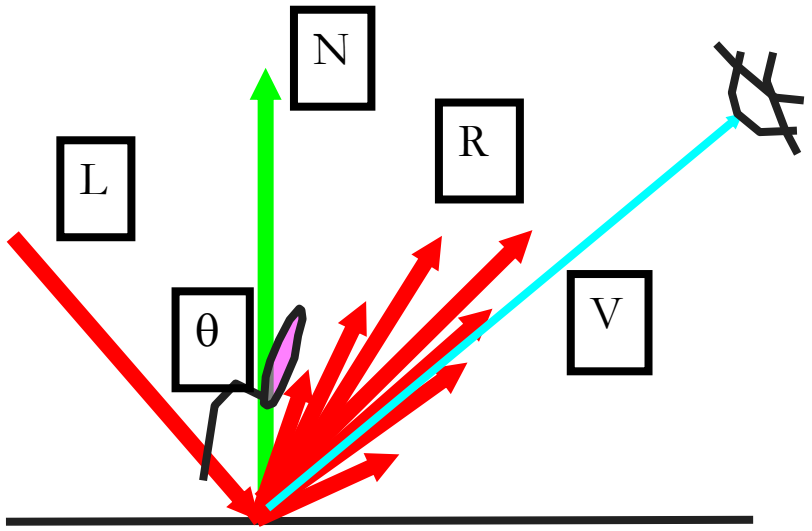
shininess = specular reflection parameter (1 -> dull, 100+ -> very shiny)

Clamping & Spotlights

$$I_{final} = I_{ambient} k_{ambient} + I_{diffuse} k_{diffuse} (N \cdot L) + I_{specular} k_{specular} (N \cdot H)^{shininess}$$

$$I_{final} = \sum_{l=0}^{lights-1} I_{l_{ambient}} k_{ambient} + I_{l_{diffuse}} k_{diffuse} (N \cdot L) + I_{l_{specular}} k_{specular} (N \cdot H)^{shininess}$$

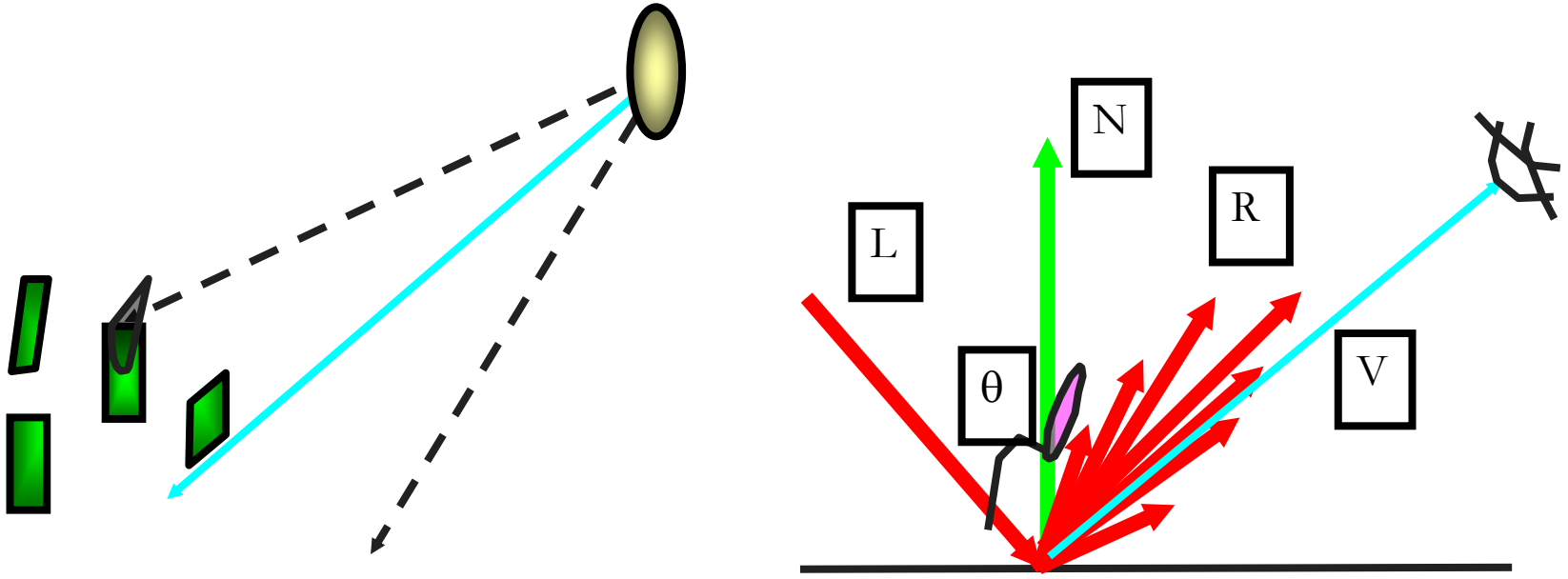
- What does the value I_{final} mean?
- How do we make sure it doesn't get too high?
- Spotlights? How do them?



How would we light a green cube?

$$I_{final} = I_{ambient} k_{ambient} + I_{diffuse} k_{diffuse} (N \cdot L) + I_{specular} k_{specular} (N \cdot H)^{shininess}$$

$$I_{final} = \sum_{l=0}^{lights-1} I_{l_{ambient}} k_{ambient} + I_{l_{diffuse}} k_{diffuse} (N \cdot L) + I_{l_{specular}} k_{specular} (N \cdot H)^{shininess}$$



Attenuation

- One factor we have yet to take into account is that a light source contributes a higher incident intensity to closer surfaces.
- The energy from a point light source falls off proportional to $1/d^2$.
- What happens if we *don't* do this?

What would attenuation do for:

- Actually, using *only* $1/d^2$, makes it difficult to correctly light things. Think if d=1 and d=2. Why?
- Remember, we are approximating things. Lighting model is too simple AND most lights are not point sources.
- We use: $$f(d) = \frac{1}{a_0 + a_1 d + a_2 d^2}$$

Subtleties

- What's wrong with:

$$f(d) = \frac{1}{a_0 + a_1 d + a_2 d^2}$$

What's a good fix?

$$f(d) = \min\left(1, \frac{1}{a_0 + a_1 d + a_2 d^2}\right)$$

Full Illumination Model

$$I_{final} = I_{l_{ambient}} k_{ambient} + \sum_{l=0}^{lights-1} f(d_l) \left[I_{l_{diffuse}} k_{diffuse} (N \cdot L) + I_{l_{specular}} k_{specular} (N \cdot H)^{shininess} \right]$$

$$f(d) = \min\left(1, \frac{1}{a_0 + a_1 d + a_2 d^2}\right)$$

Run demo

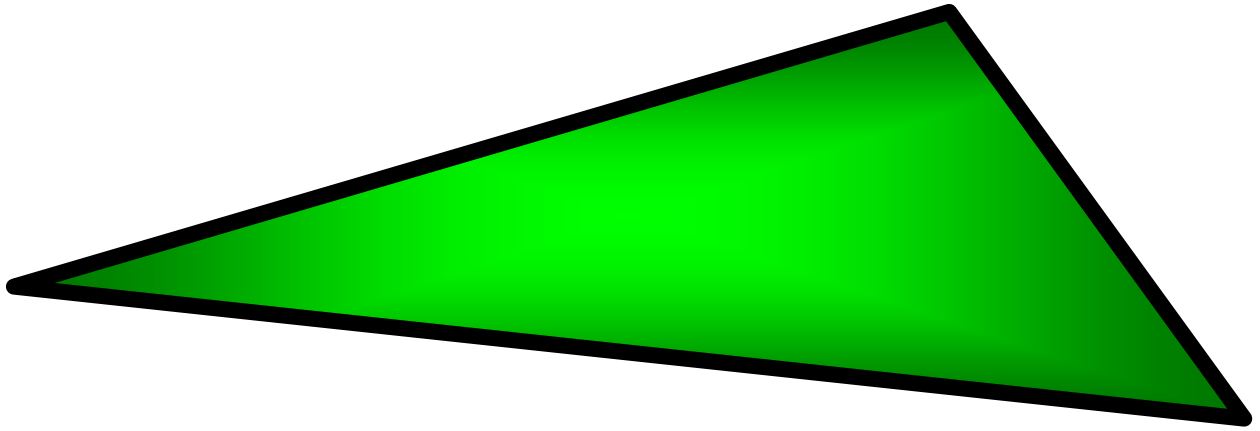
Putting Lights in OpenGL

- 1. glEnable(GL_LIGHTING);
- 2. Set up Light properties
  - glLightf(…)
- 3. Set up Material properties
  - glMaterial(…)

Shading

$$I_{final} = I_{l_{ambient}} k_{ambient} + \sum_{l=0}^{lights-1} f(d_l) \left[I_{l_{diffuse}} k_{diffuse} (N \cdot L) + I_{l_{specular}} k_{specular} (N \cdot H)^{shininess} \right]$$

- When do we do the lighting equation?
- What is the cost to compute the lighting for a 3D point?



Shading

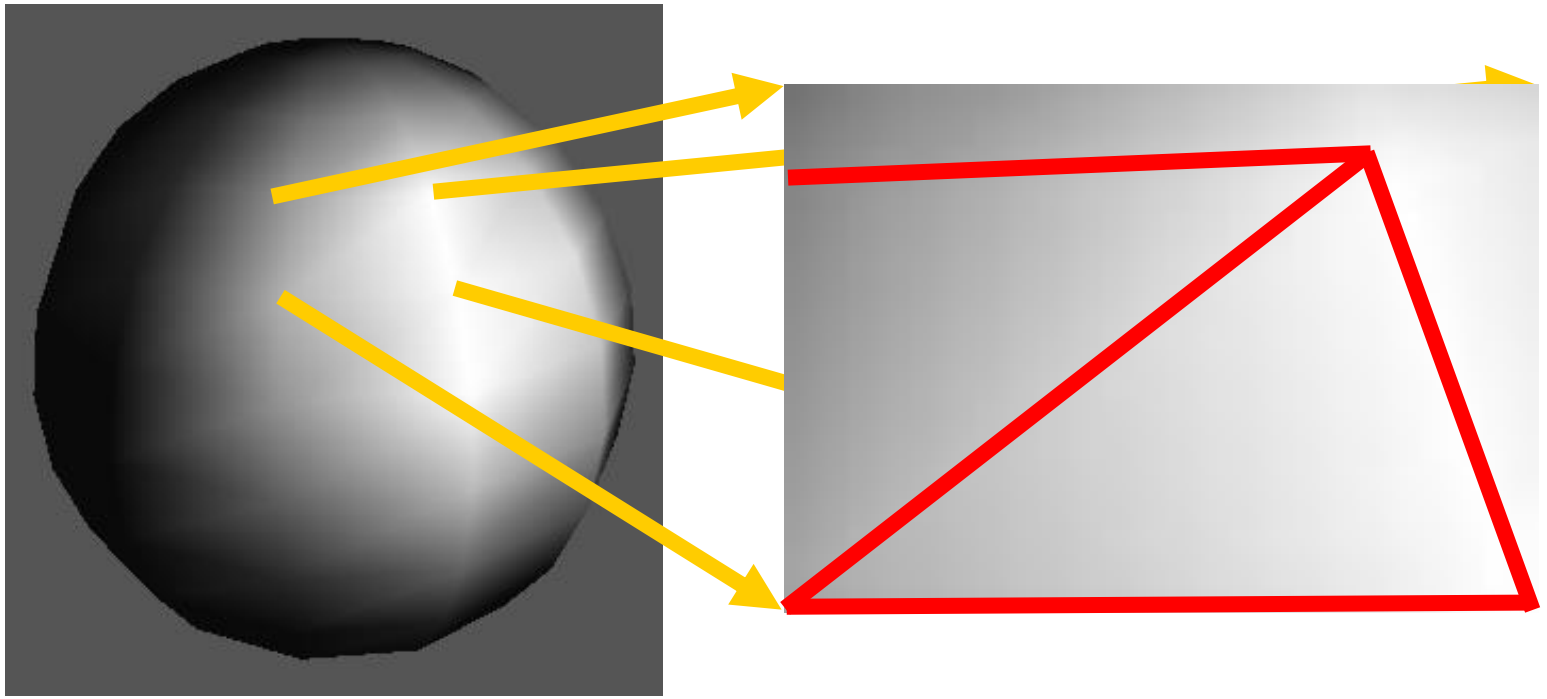
- Shading is how we “color” a triangle.
- Constant Shading
- Gouraud Shading
- Phong Shading

Constant Shading

- Constant Intensity or Flat Shading
- One color for the entire triangle
- Fast
- Good for some objects
- What happens if triangles are small?
- Sudden intensity changes at borders

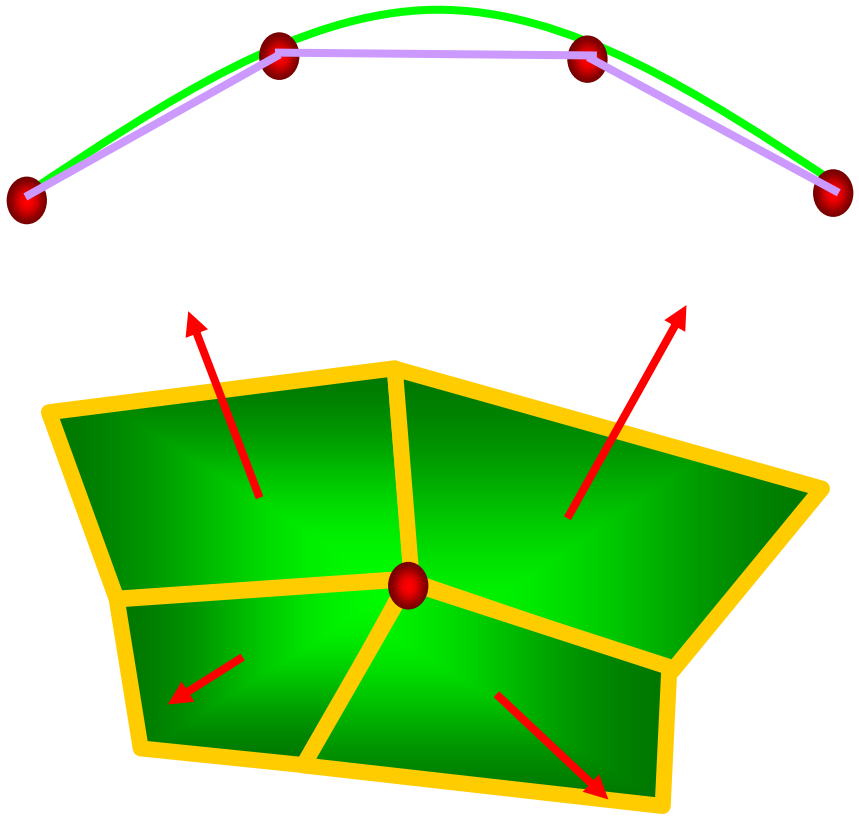
Gouraud Shading

- Intensity Interpolation Shading
- Calculate lighting at the vertices. Then interpolate the colors as you scan convert



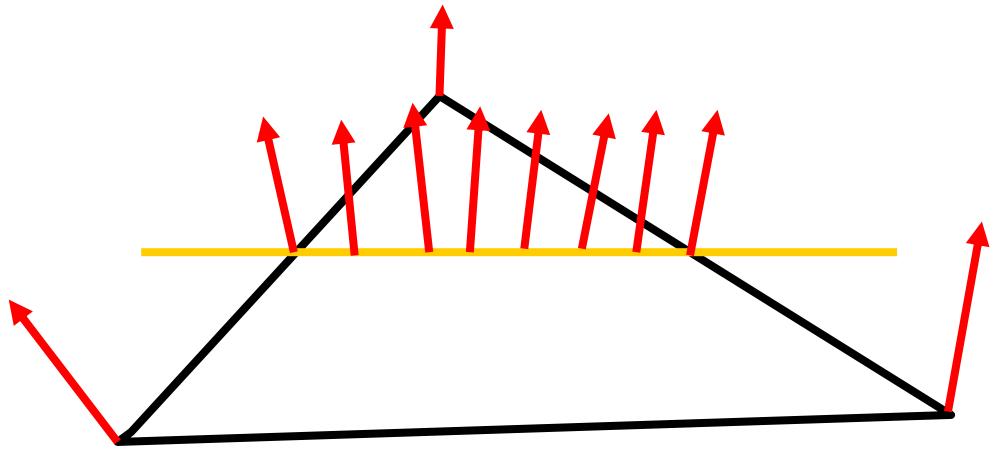
Gouraud Shading

- Relatively fast, only do three calculations
- No sudden intensity changes
- What can it not do?
- What are some approaches to fix this?
- Question, what is the normal at a vertex?



Phong Shading

- Interpolate the normal, since that is the information that represents the "curvature"
- Linearly interpolate the vertex normals. For ***each*** pixel, as you scan convert, *calculate* the lighting *per pixel*.
- True "per pixel" lighting
- Not done by most hardware/libraries/etc



Shading Techniques

- Constant Shading
  - Calculate one lighting calculation (pick a vertex) per triangle
  - Color the *entire* triangle the same color
- Gouraud Shading
  - Calculate three lighting calculations (the vertices) per triangle
  - Linearly interpolate the colors as you scan convert
- Phong Shading
  - While you scan convert, linearly interpolate the normals.
  - With the interpolated normal at each pixel, calculate the lighting at each pixel

Color Spectrum

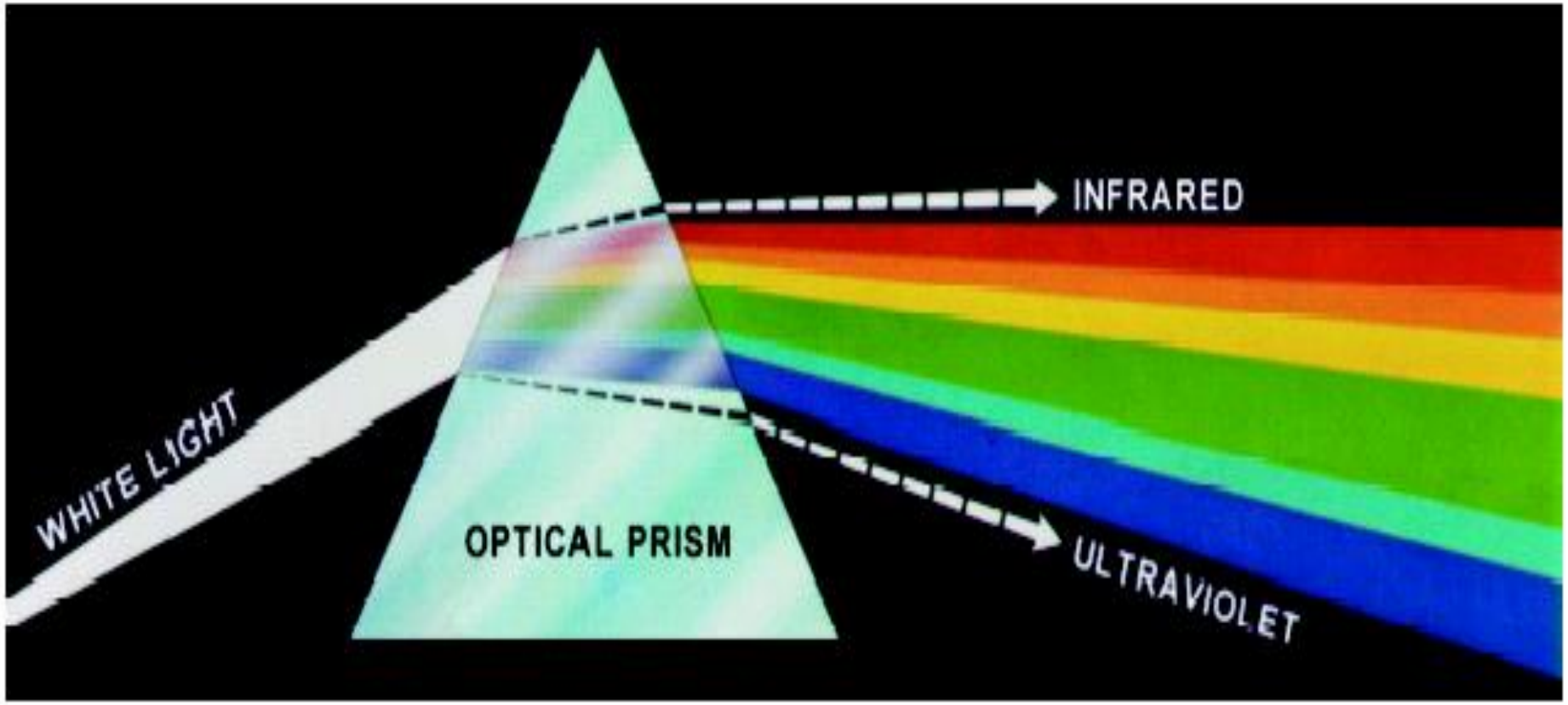


FIGURE 6.1 Color spectrum seen by passing white light through a prism. (Courtesy of the General Electric Co., Lamp Business Division.)

Electromagnetic Spectrum

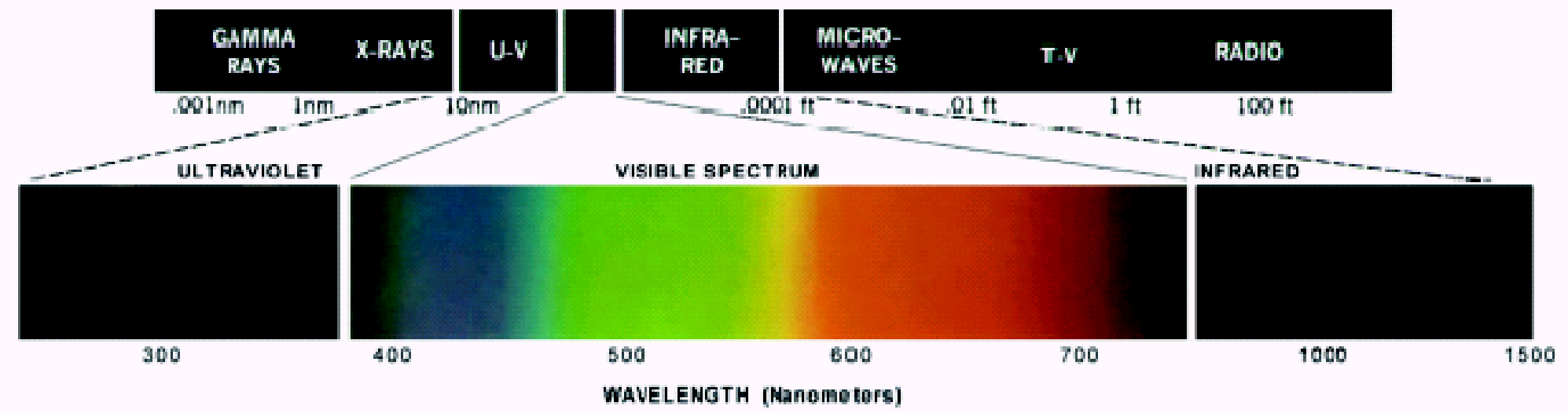
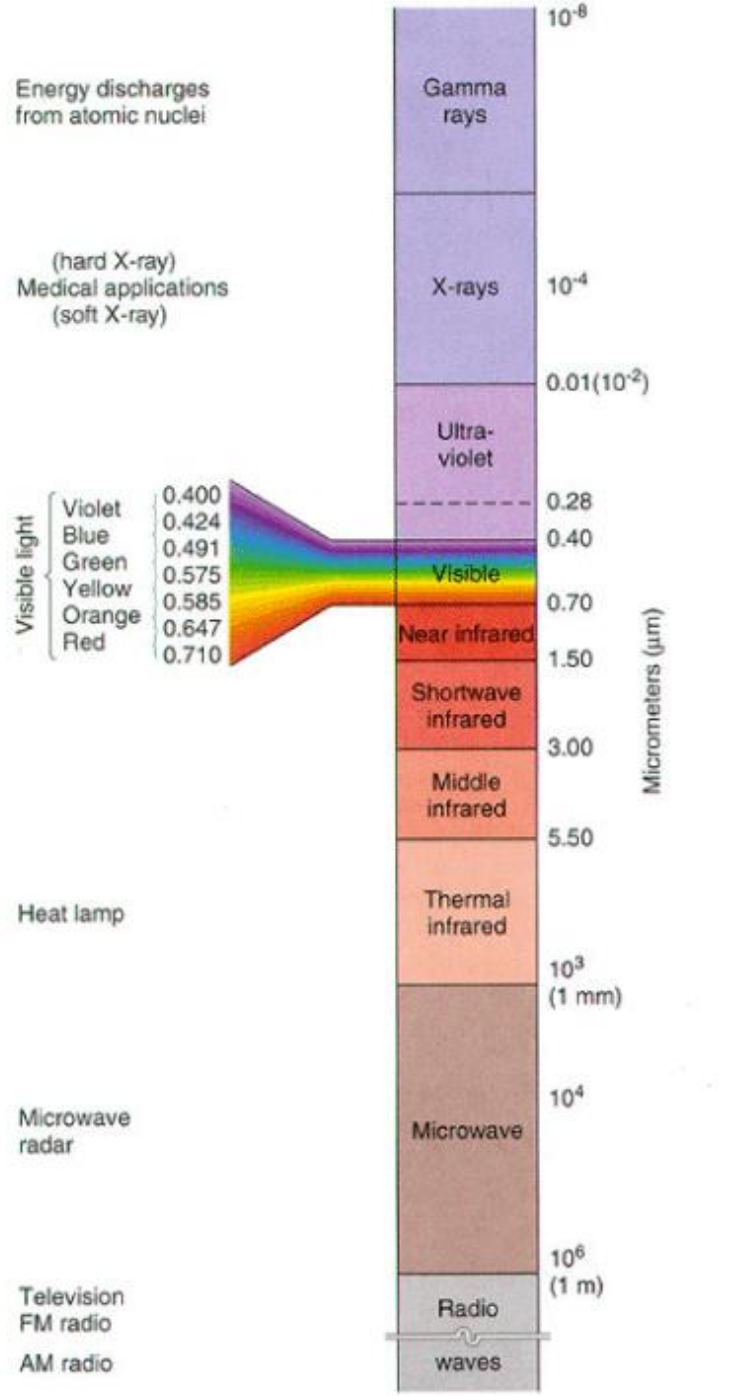


FIGURE 6.2 Wavelengths comprising the visible range of the electromagnetic spectrum. (Courtesy of the General Electric Co., Lamp Business Division.)

Physical Background

- Visible light: a narrow band of electromagnetic radiation → **380nm (blue) - 780nm (red)**

- Wavelength: Each physically distinct colour corresponds to at least one wavelength in this band.



source: Christopherson (2000) *Geosystems*

Color Fundamentals

- The colors that humans and some animals perceive in an object are determined by the nature of light reflected from the object

Achromatic vs Chromatic Light

- Achromatic (void of color) Light: Its only contribute is its 'Intensity' or amount

- Chromatic Light: spans the electromagnetic spectrum from approximately 400 to 700nm

Human Perception

- Detailed experimental evidences has established that the 6 to 7 million cones in the human eye can be divided into three principal sensing categories, corresponding roughly to red, green and blue

- Approximately 65% of all cones are sensitive to Red Light, 33% are sensitive to Green Light and about 2% are sensitive to Blue Light (most sensitive)

Human Perception

- Due to these absorption characteristic of Human Eye colors are seen as variable combinations of the so-called 'Primary Colors' Red, Green and Blue

- The primary colors can be added to produce secondary colors of Light

  - Magenta (Red+Blue)
  - Cyan (Green+Blue)
  - Yellow (Red+Green)

Absorption of Light by red, green and blue cones in Human Eye

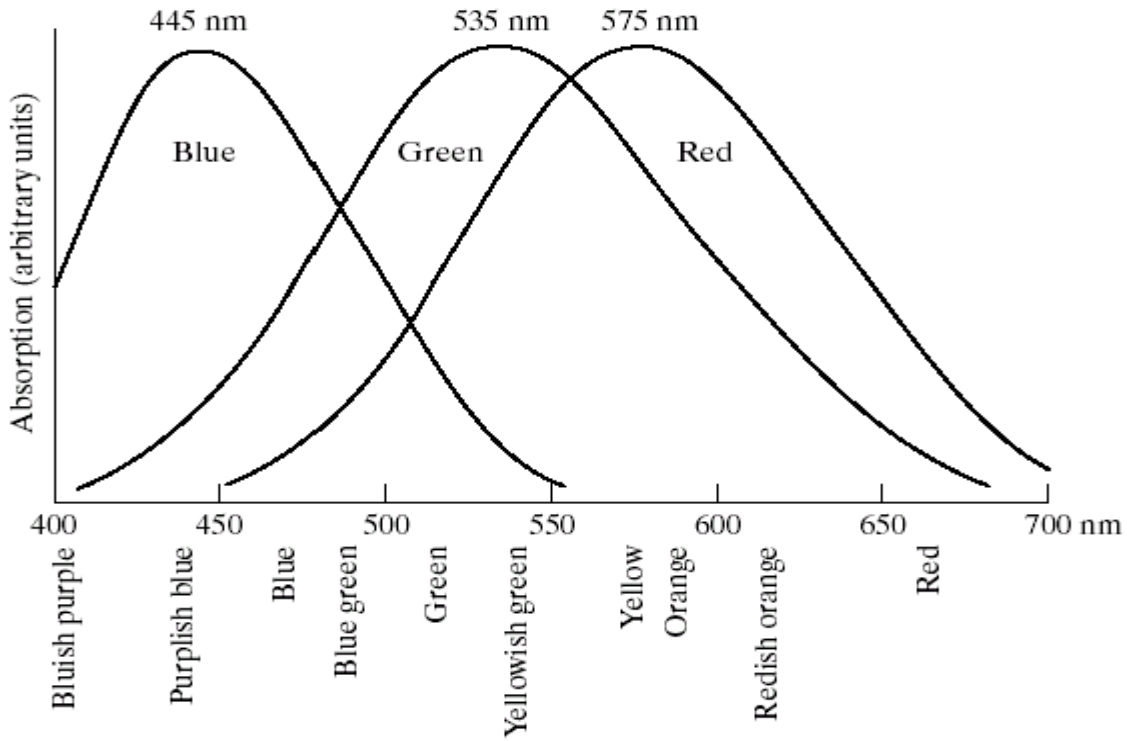
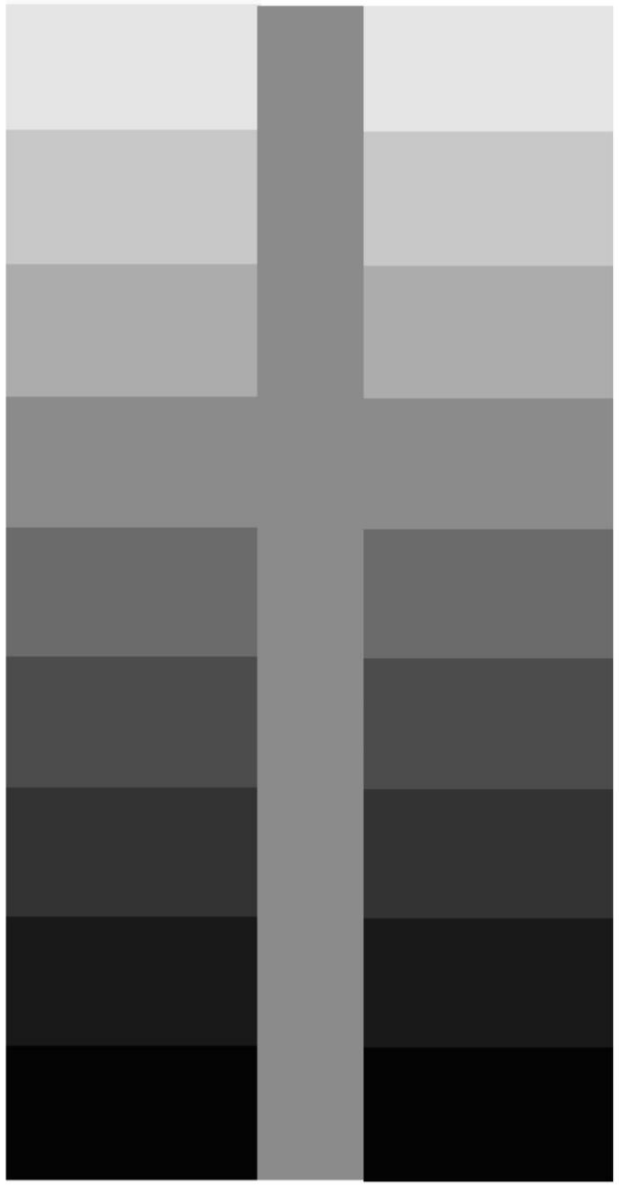


FIGURE 6.3 Absorption of light by the red, green, and blue cones in the human eye as a function of wavelength.

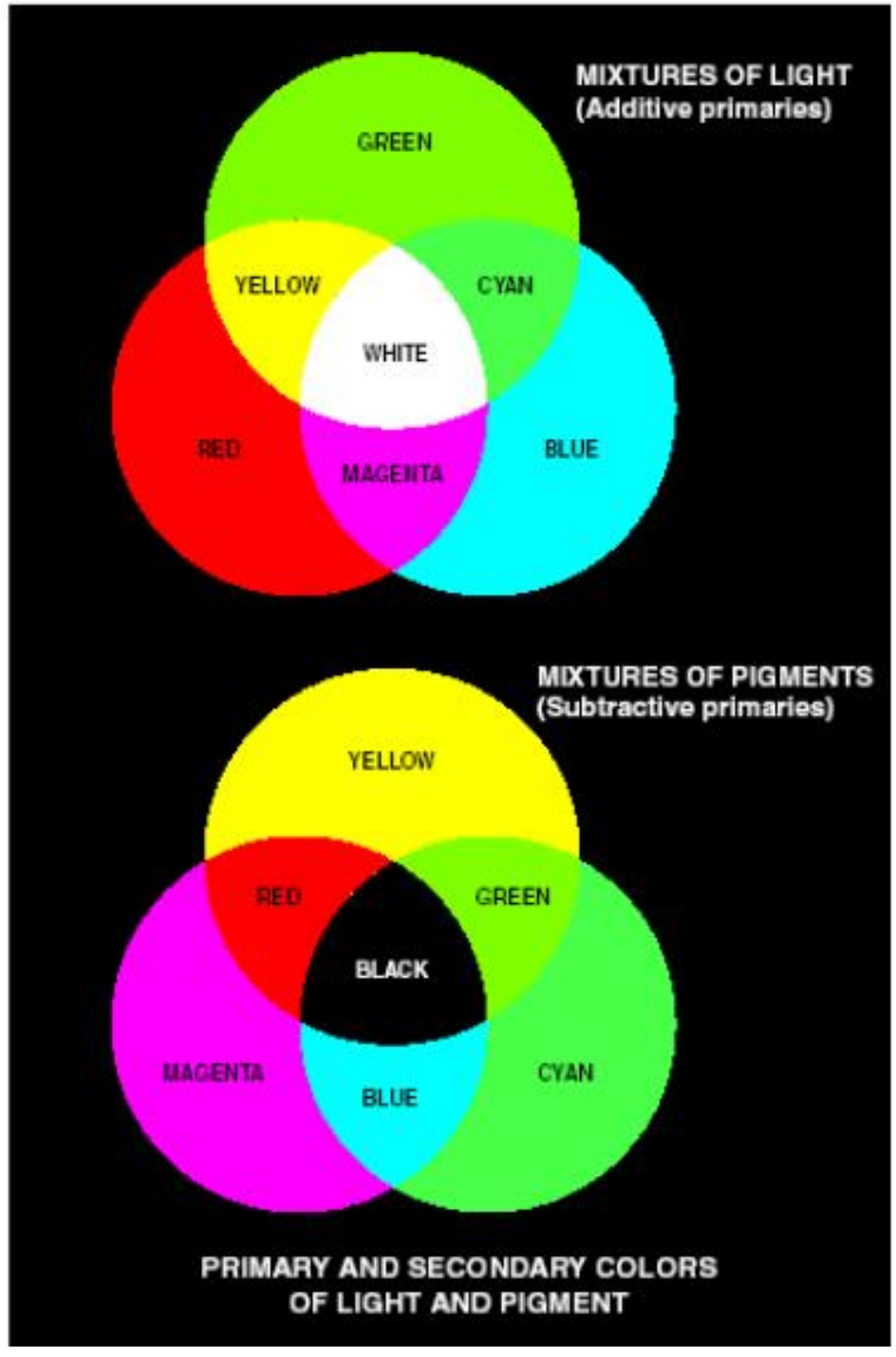
- Mixing the three primaries or a secondary with its opposite primary colors in the right intensities produces white light

Brightness



Primary Color of Light vs Primary Color of Pigments

- Red, Green and Blue Colors are Primary Colors of Light
- In Primary Color of Pigments a primary color is defined as the one that subtracts or absorbs a primary color of Light and reflects or transmits the other two
- Therefore the Primary Colors of Pigments are Magenta, Cyan and Yellow and secondary colors are Red, Green and Blue
- A proper combination of three pigment primaries or a secondary with its opposite primary produces Black
- Color Television Reception is an example of the additive nature of Light Colors


MIXTURES OF LIGHT
(Additive primaries)
GREEN
YELLOW
CYAN
WHITE
RED
MAGENTA
BLUE
MIXTURES OF PIGMENTS
(Subtractive primaries)
YELLOW
RED
GREEN
BLACK
MAGENTA
BLUE
CYAN
PRIMARY AND SECONDARY COLORS
OF LIGHT AND PIGMENT

Color Models

- The purpose of a color model (also called Color Space or Color System) is to facilitate the specification of colors in some standard way

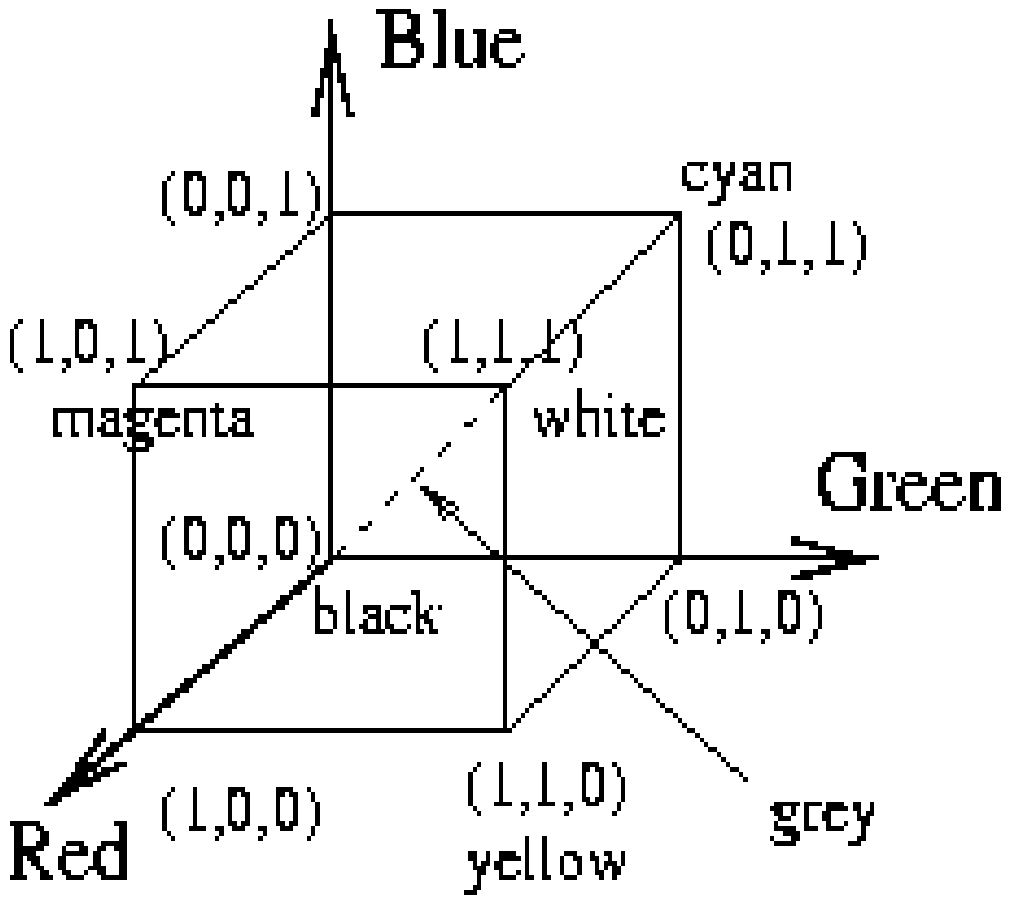
- A color model is a specification of a coordinate system and a subspace within that system where each color is represented by a single point

- **Color Models**

  RGB (Red, Green, Blue)
  CMY (Cyan, Magenta, Yellow)
  HSI (Hue, Saturation, Intensity)
  YIQ (Luminance, In phase, Quadrature)
  YUV (Y' stands for the luma component (the brightness) and U and V are the chrominance (color) components)

RGB Model

- Each color is represented in its primary color components Red, Green and Blue
- This model is based on Cartesian Coordinate System



RGB Model

- In this model, the primary colors are red, green, and blue. It is an additive model, in which colors are produced by adding components, with white having all colors present and black being the absence of any color.

- This is the model used for active displays such as television and computer screens.

- The RGB model is usually represented by a unit cube with one corner located at the origin of a three-dimensional color coordinate system, the axes being labeled R, G, B, and having a range of values [0, 1]. The origin (0, 0, 0) is considered black and the diagonally opposite corner (1, 1, 1) is called white. The line joining black to white represents a gray scale and has equal components of R, G, B.

RGB Color Cube

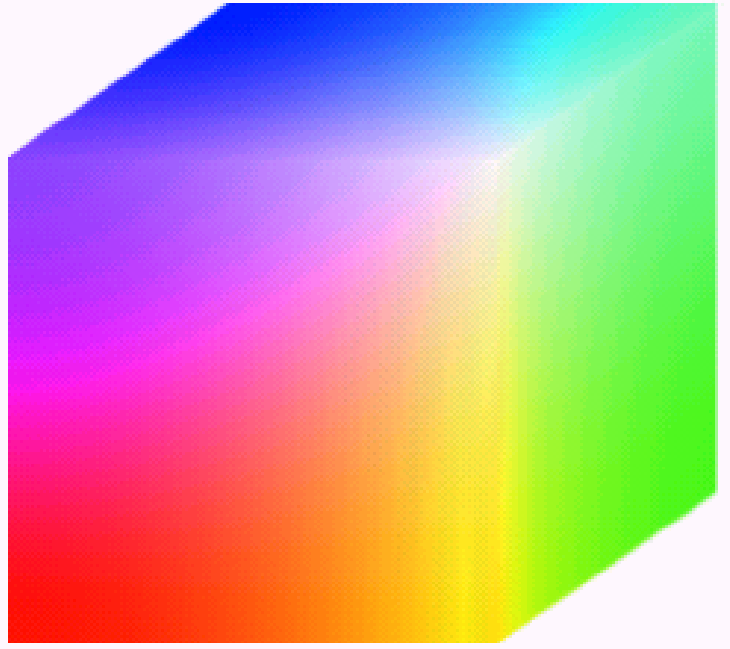


FIGURE 6.8 RGB 24-bit color cube.

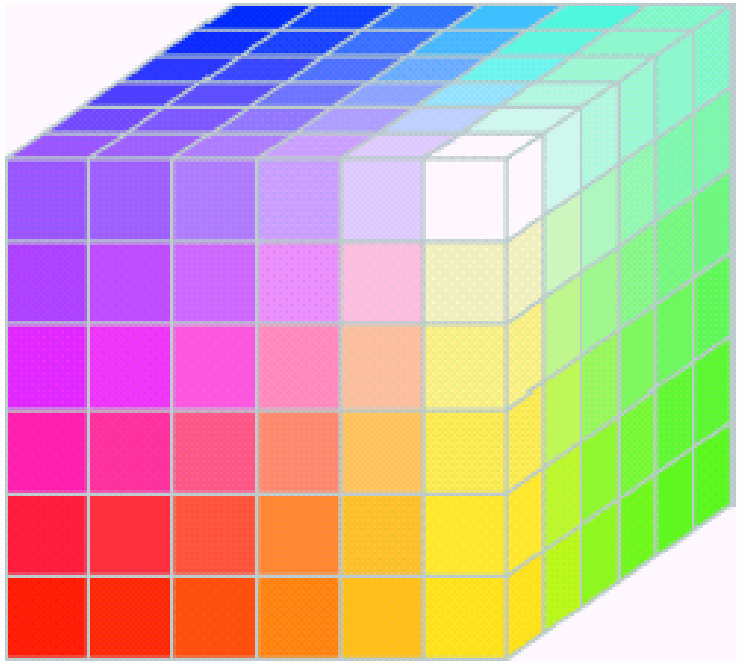


FIGURE 6.11 The RGB safe-color cube.

- The total number of colors in a 24 Bit image is $(2^8)^3$ =16,777,216 (> 16 million)

CMY and CMYK Color Model

- Cyan, magenta, and yellow are the secondary colors with respect to the primary colors of red, green, and blue. However, in this subtractive model, they are the primary colors and red, green, and blue, are the secondaries. In this model, colors are formed by subtraction, where adding different pigments causes various colors not to be reflected and thus not to be seen. Here, white is the absence of colors, and black is the sum of all of them. This is generally the model used for printing.

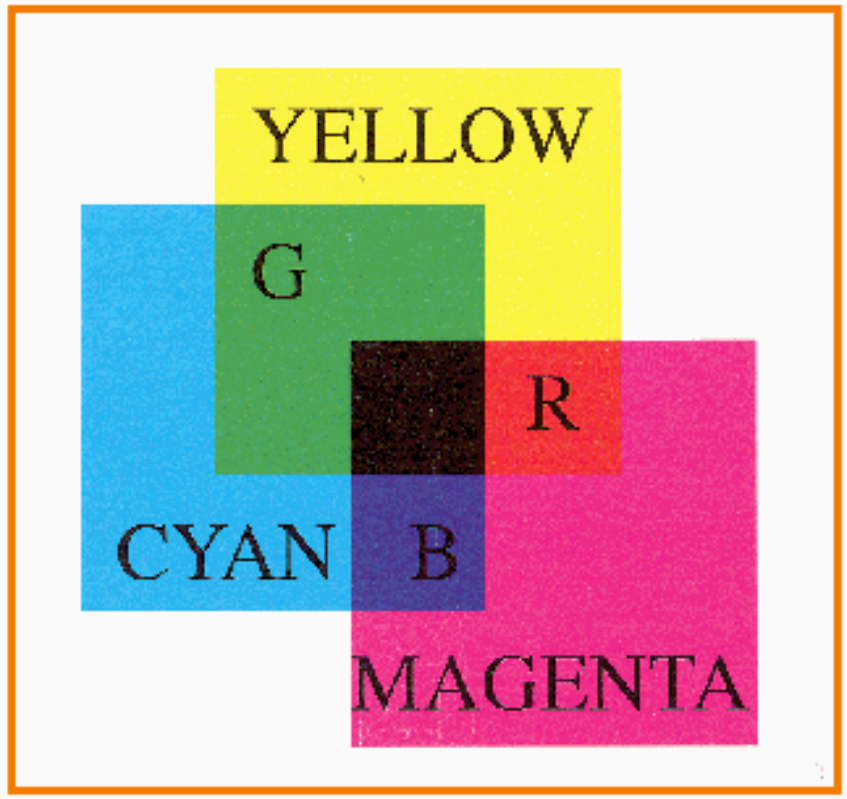
- Most devices that deposit color pigments on paper (such as Color Printers and Copiers) requires CMY data input or perform RGB to CMY conversion internally

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1.00 \\ 1.00 \\ 1.00 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

CMY and CMYK Color Model

- CMY is a Subtractive Color Model
- Equal amounts of Pigment primaries (Cyan, Magenta and Yellow) should produce Black
- In practice combining these colors for printing produces a "Muddy-Black" color
- So in order to produce "True-Black" a fourth color "Black" is added giving rise to CMYK model

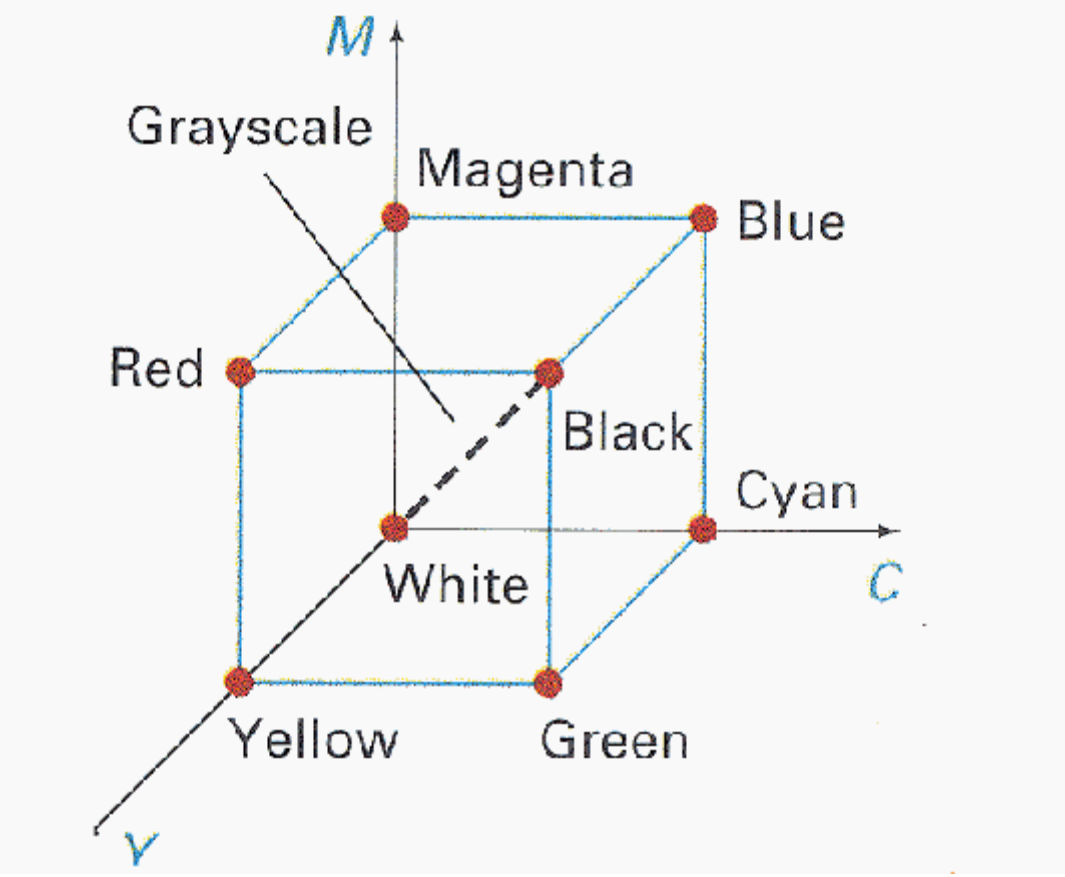
CMY Color Model



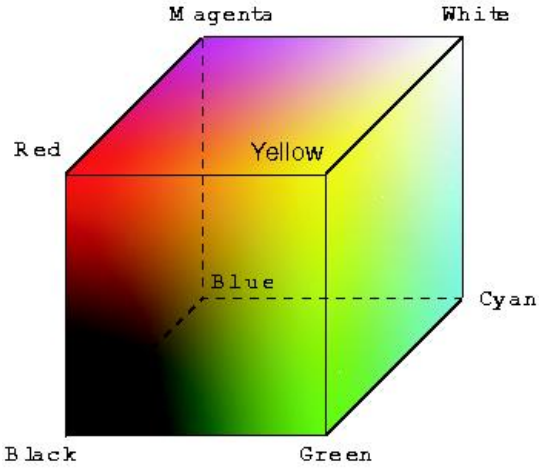
Colors are subtractive

C	M	Y	Color
0.0	0.0	0.0	White
1.0	0.0	0.0	Cyan
0.0	1.0	0.0	Magenta
0.0	0.0	1.0	Yellow
1.0	1.0	0.0	Blue
1.0	0.0	1.0	Green
0.0	1.0	1.0	Red
1.0	1.0	1.0	Black
0.5	0.0	0.0	
1.0	0.5	0.5	
1.0	0.5	0.0	

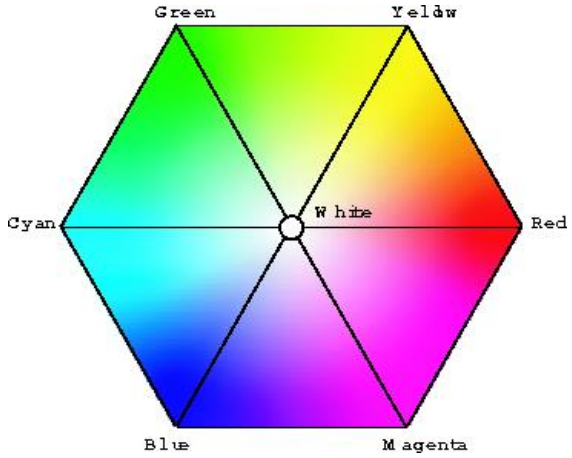
CMY Color Model



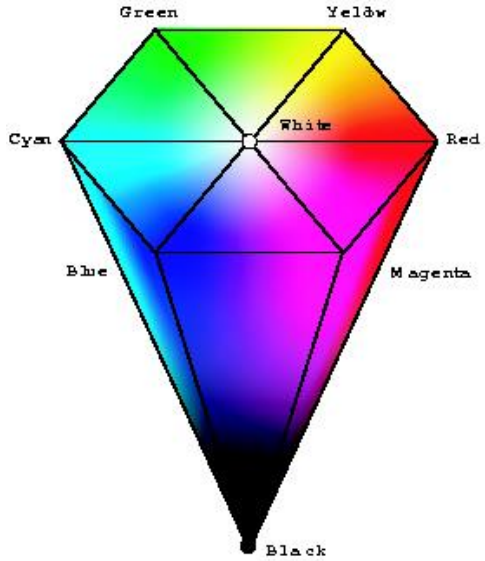
HSV Color Model



RGB cube



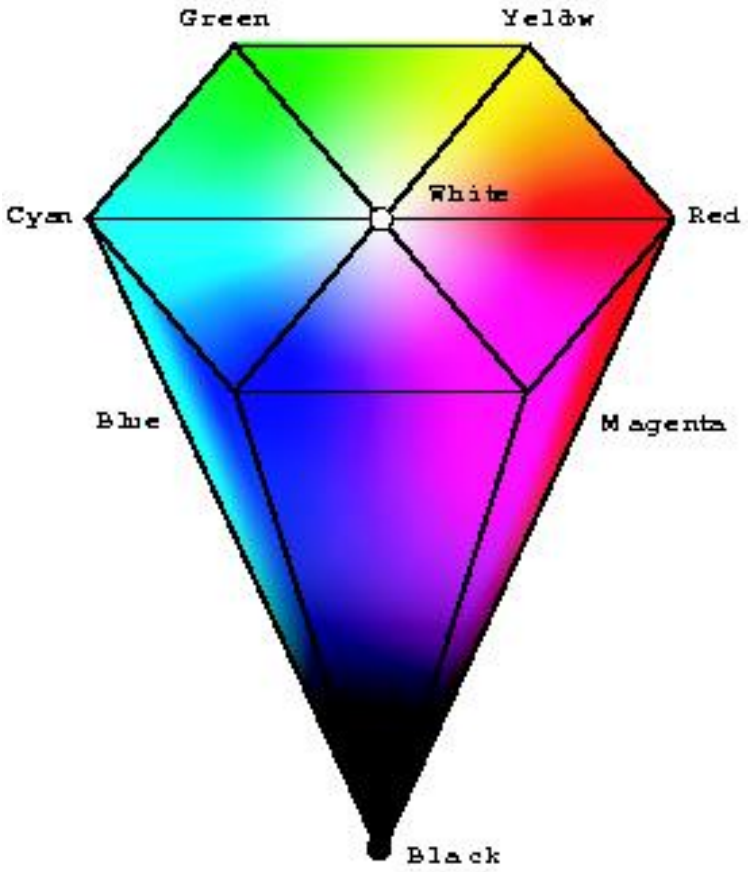
HSV top view

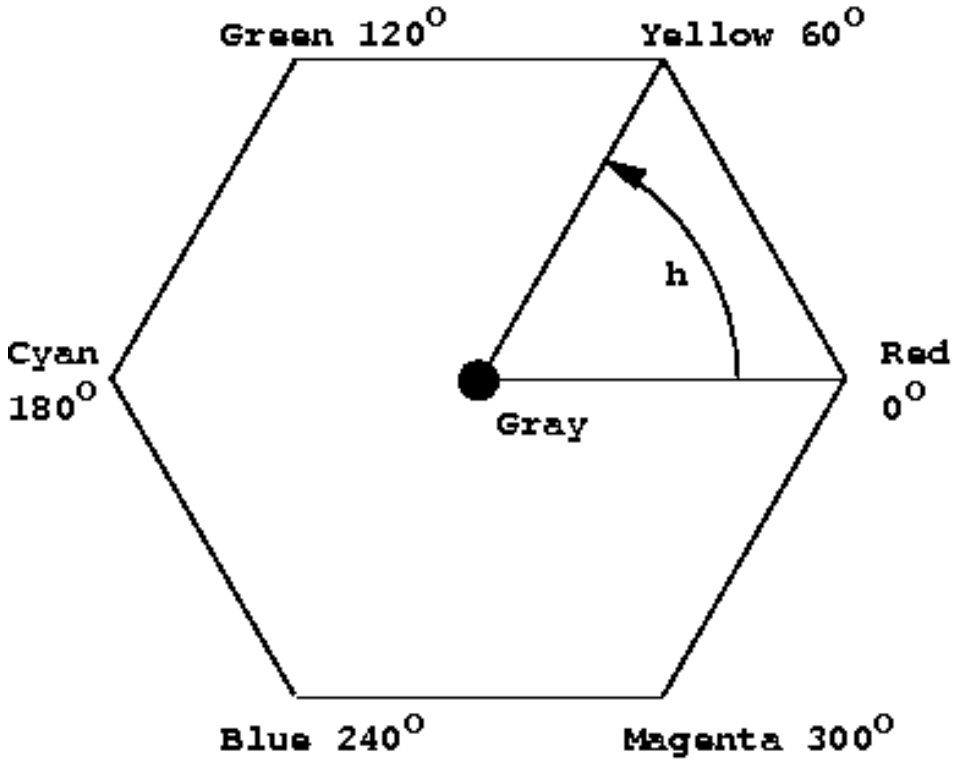


HSV cone

HSV is a projection of the RGB space

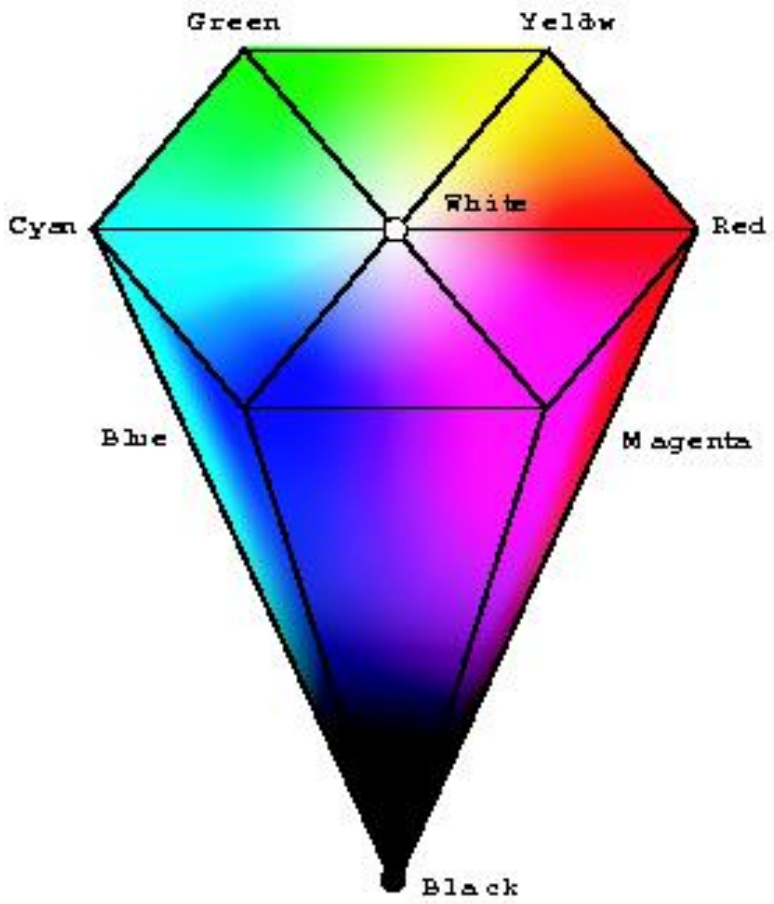
HSV Color Model

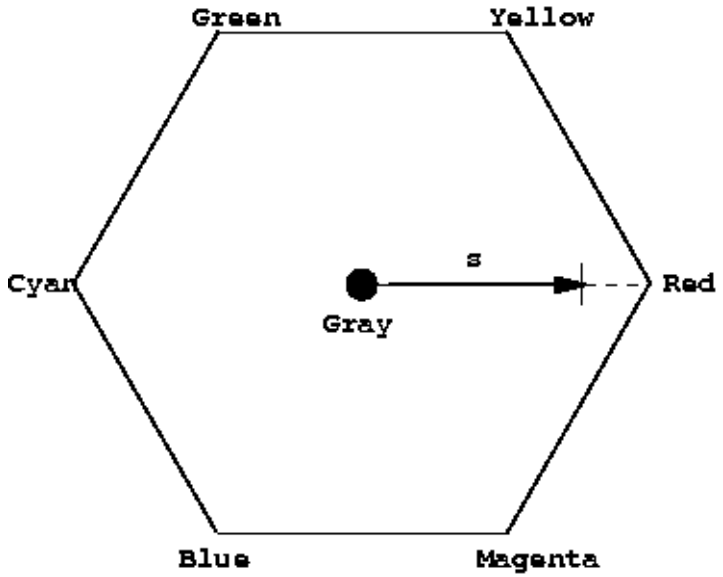




, an angular measure (0 … 360)

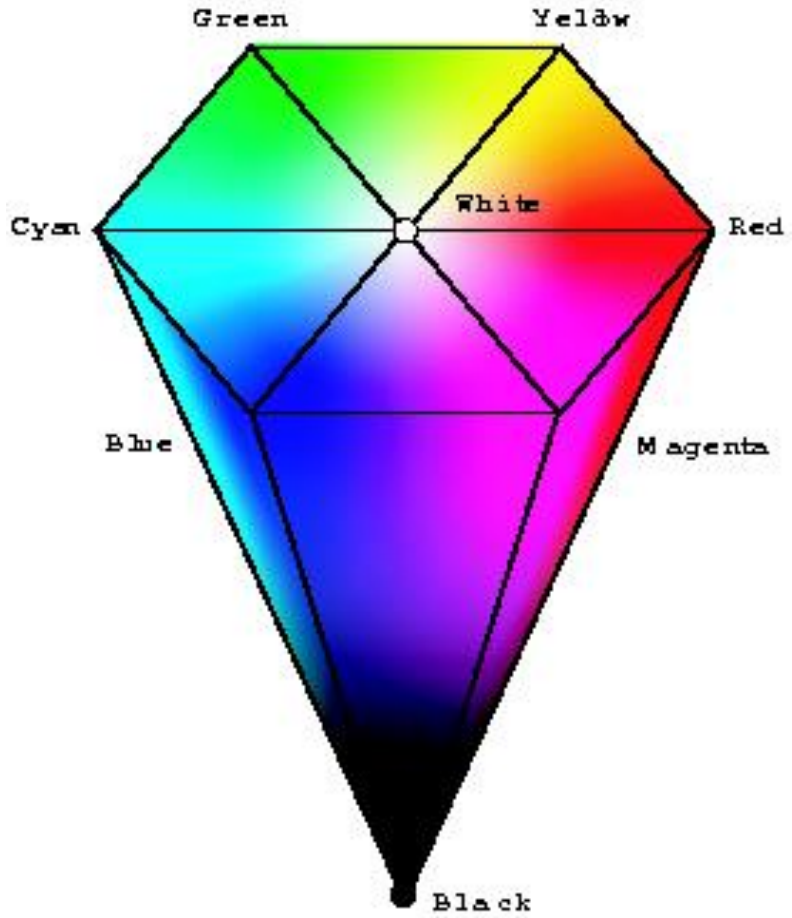
HSV Color Model

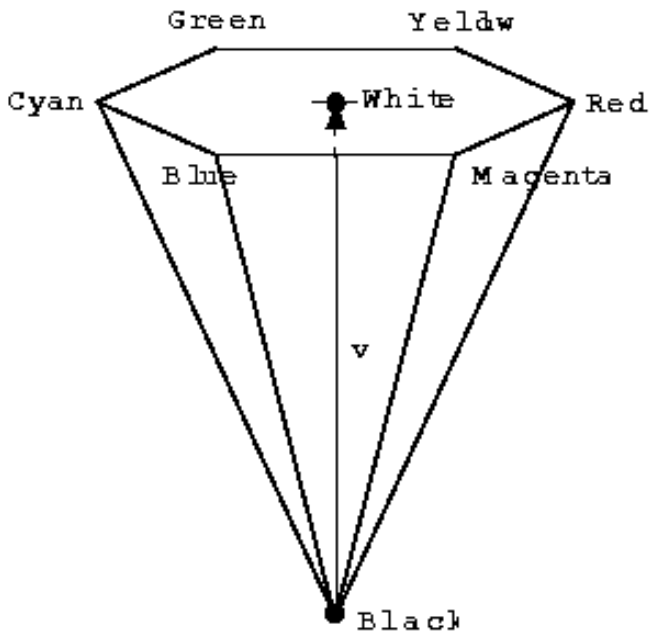




Saturation, a fractional measure (0.0 … 1.0)

HSV Color Model

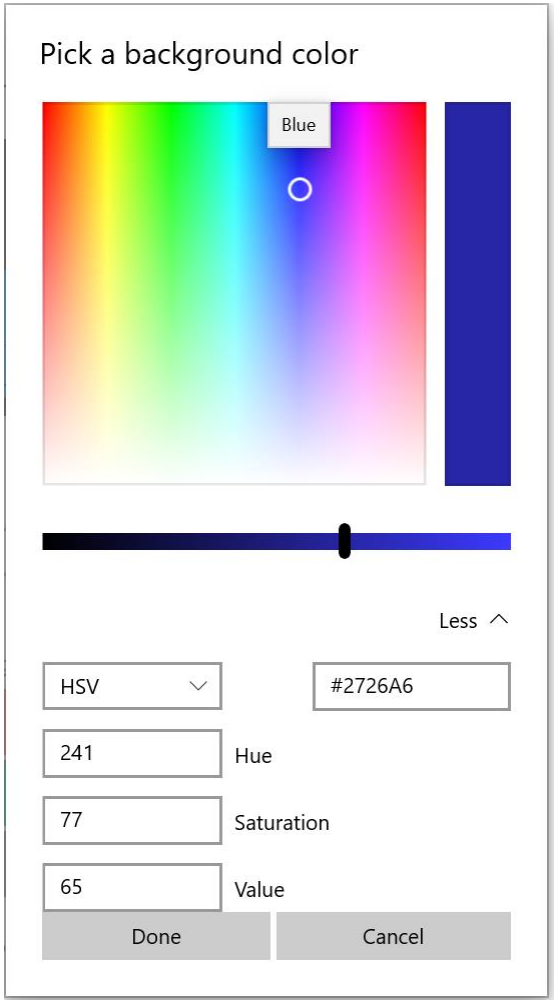




Value, a fractional measure (0.0 … 1.0)

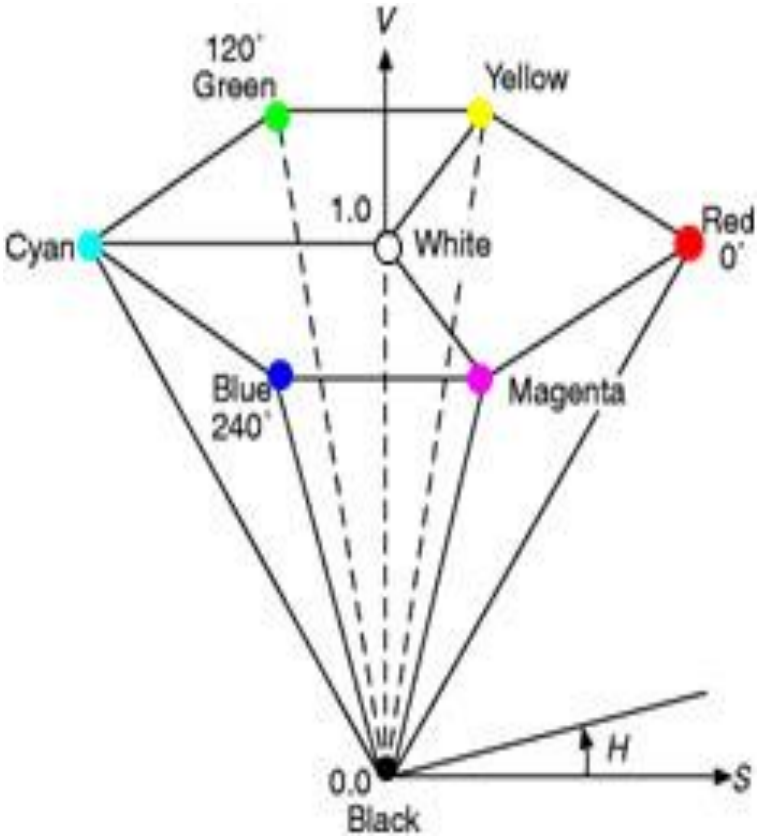
Color Model of Light

Additive Color



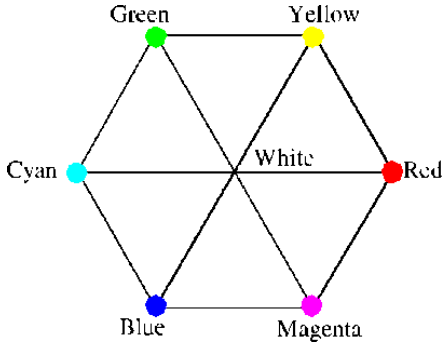
The HSV Color Model

- Hue, Saturation, Value (Brightness)
- HSV-space invented by Alvy Ray Smith—described in his 1978 SIGGRAPH paper, *Color Gamut Transformation Pairs.*
- Hexcone subset of cylindrical (polar) coordinate system
- Single hexcone HSV color model. The cross-section at V = 1 contains the RGB model's R =1, G = 1, B = 1

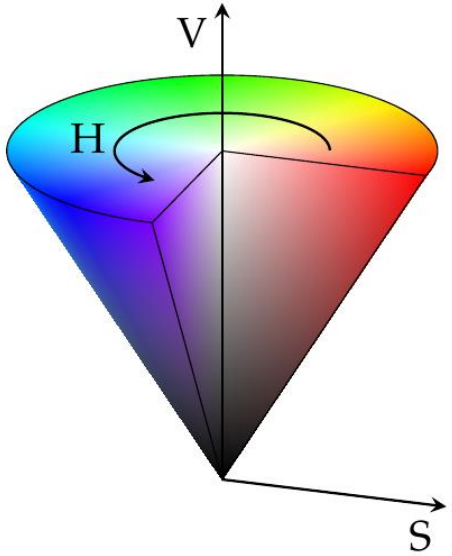


The HSV Color Model

- Colors on V = 1 plane are not equally bright perceptually
- Complementary colors 180° opposite (Hue = 0 to 360^0)
- Saturation measured relative to color gamut represented by mathematical HSV model which is subset of the perceptually-based chromaticity diagram for a given value (Color Button in TV):
- Top of HSV hexcone is projection seen by looking along principal diagonal of RGB color
- RGB subcubes are plane of constant V
- Note: linear path RGB ≠ linear path in HSV! (has consequences for interpolation/animation)



RGB color cube viewed along the principal diagonal



The HSV Color Model

- RGB to HSV conversion

$C_{max} = max(R,G,B);$

$\Delta = C_{max} - min(R,G,B);$

$V = C_{max};$

if (V == 0){

$S = 0;$

}

else {

$S = \Delta / C_{max};$

}

if (S == 0){

$H = undefined;$

}

else {

if (C_{max}== R){

$$H = \left(\frac{G-B}{\Delta} \times 60\right) mod\ 360;$$

}

else if (C_{max}== G){

$$H = \left(\frac{B-R}{\Delta} \times 60\right) + 120;$$

}

else {

$$H = \left(\frac{R-G}{\Delta} \times 60\right) + 240;$$

}

}

The HSV Color Model

- HSV to RGB conversion

$$C_1 = V \times S$$

$$C_2 = C_1 \times \left(1 - \left|\frac{H}{60^\circ} \bmod 2 - 1\right|\right)$$

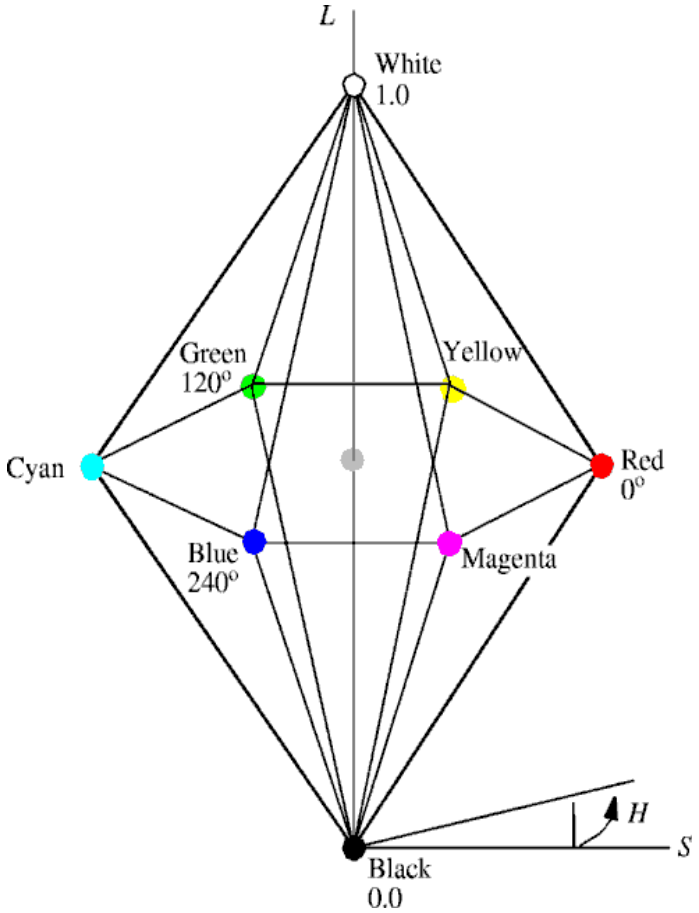
$$(R', G', B') = \begin{cases} (C_1, C_2, 0), & 0^\circ < H < 60^\circ \\ (C_2, C_1, 0), & 60^\circ < H < 120^\circ \\ (0, C_1, C_2), & 120^\circ < H < 180^\circ \\ (0, C_2, C_1), & 180^\circ < H < 240^\circ \\ (C_2, 0, C_1), & 240^\circ < H < 300^\circ \\ (C_1, 0, C_2), & 300^\circ < H < 360^\circ \end{cases}$$

$$C_{min} = V - C_1$$

$$(R, G, B) = (R' + C_{min}, \; G' + C_{min}, \; B' + C_{min})$$

The HLS Color Model

- Hue, Lightness, Saturation
- Double-hexcone subset of XYZ space
- Maximally saturated hues are at S = 1, L = 0.5
- Conceptually easier for some people to view white as a point



The HLS Color Model

- RGB to HLS conversion

$C_{max} = max(R,G,B);$

$C_{min} = min(R,G,B);$

$\Delta = C_{max} - C_{min};$

$L = (C_{max} + C_{min})/2;$

if (L == 0){

$S = 0;$

}

else {

$$S = \frac{\Delta}{1 - |2L-1|};$$

}

if (S == 0){

$H = undefined;$

}

else {

if (C_{max} == *R){*

$$H = \left(\frac{G-B}{\Delta} \times 60\right) mod\ 360;$$

}

else if (C_{max} == *G){*

$$H = \left(\frac{B-R}{\Delta} \times 60\right) + 120;$$

}

else {

$$H = \left(\frac{R-G}{\Delta} \times 60\right) + 240;$$

}

}

The HLS Color Model

HSL to RGB conversion formula

When $0 \le H < 360$, $0 \le S \le 1$ and $0 \le L \le 1$:

$$C = (1 - |2L - 1|) \times S$$

$$X = C \times (1 - |(H / 60°) \bmod 2 - 1|)$$

$$m = L - C/2$$

$$(R', G', B') = \begin{cases} (C, X, 0) & , 0° \le H < 60° \\ (X, C, 0) & , 60° \le H < 120° \\ (0, C, X) & , 120° \le H < 180° \\ (0, X, C) & , 180° \le H < 240° \\ (X, 0, C) & , 240° \le H < 300° \\ (C, 0, X) & , 300° \le H < 360° \end{cases}$$

$$(R,G,B) = ((R'+m)\times 255, (G'+m)\times 255, (B'+m)\times 255)$$

Polygon Filling Algorithms

Polygon Representation

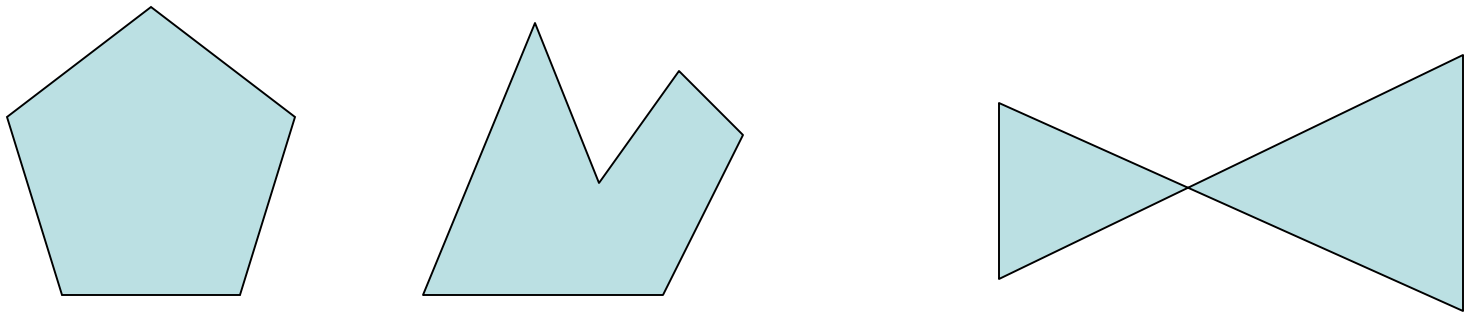
The polygon can be represented by listing its n vertices in an ordered list.

$$P = \{(x_1, y_1), (x_2, y_2), \ \ldots\ldots, (x_n, y_n)\}.$$

The polygon can be displayed by drawing a line between (x_1, y_1), and (x_2, y_2), then a line between (x_2, y_2), and (x_3, y_3), and so on until the end vertex. In order to close up the polygon, a line between (x_n, y_n), and (x_1, y_1) must be drawn.

Polygon Fill Algorithm

- ***Different types of Polygons***
  - Simple Convex
  - Simple Concave
  - Non-simple : self-intersecting
  - With holes



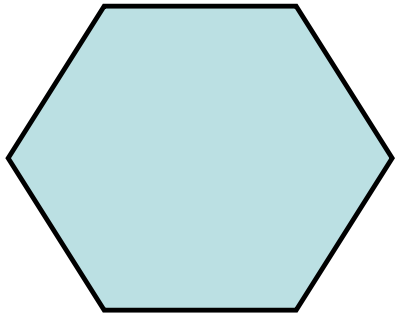
Convex Concave Self-intersecting

Polygon Filling

Types of filling

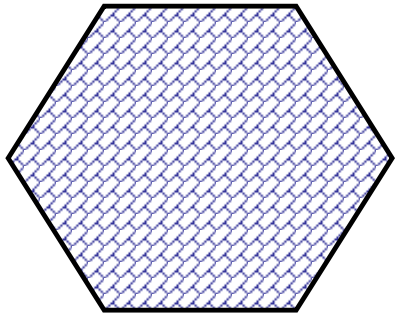
- **Solid-fill**

All the pixels inside the polygon's boundary are illuminated.



- **Pattern-fill**

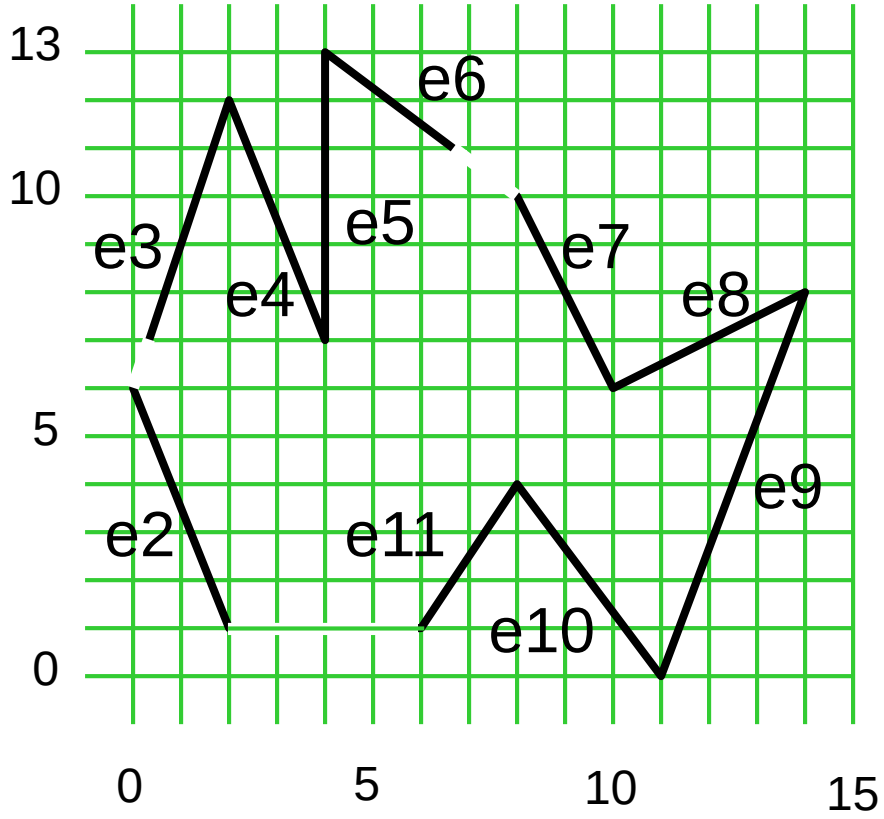
the polygon is filled with an arbitrary predefined pattern.



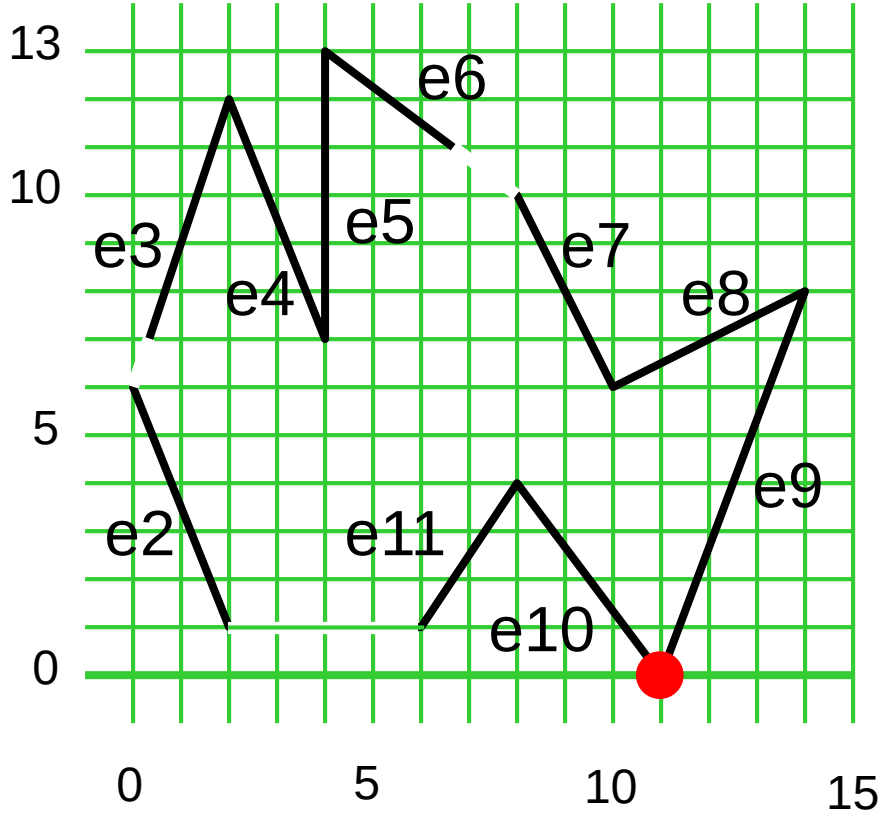
Types of Polygon fill algorithm

- 1. Scan line algorithm
- 2. Boundary Fill algorithm
- 3. Flood fill algorithm

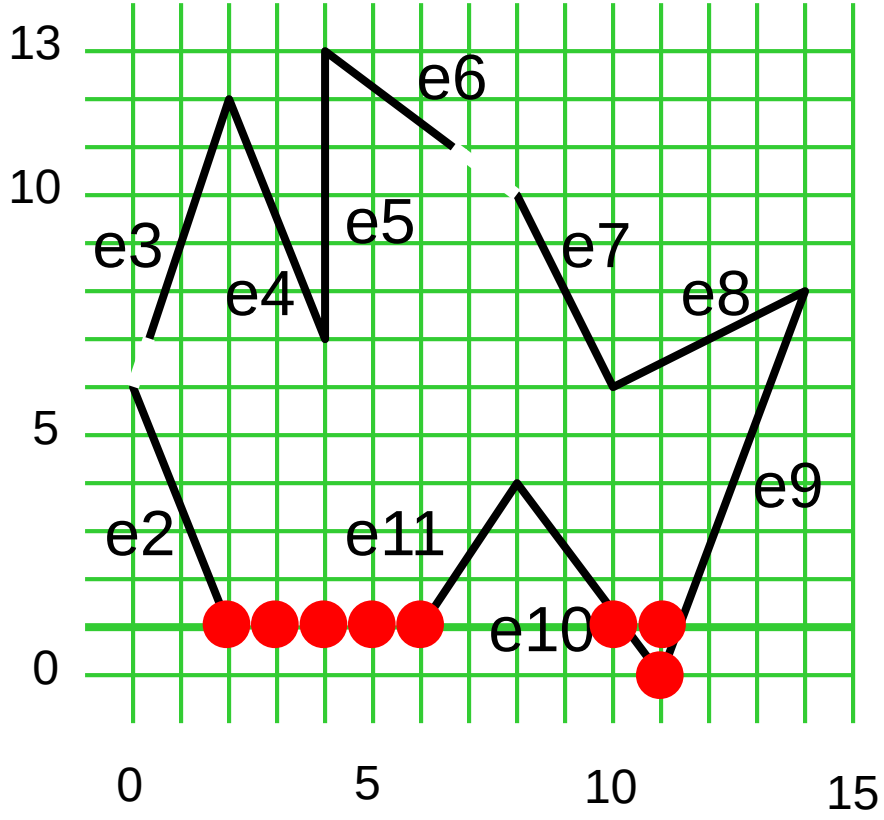
Running the Algorithm



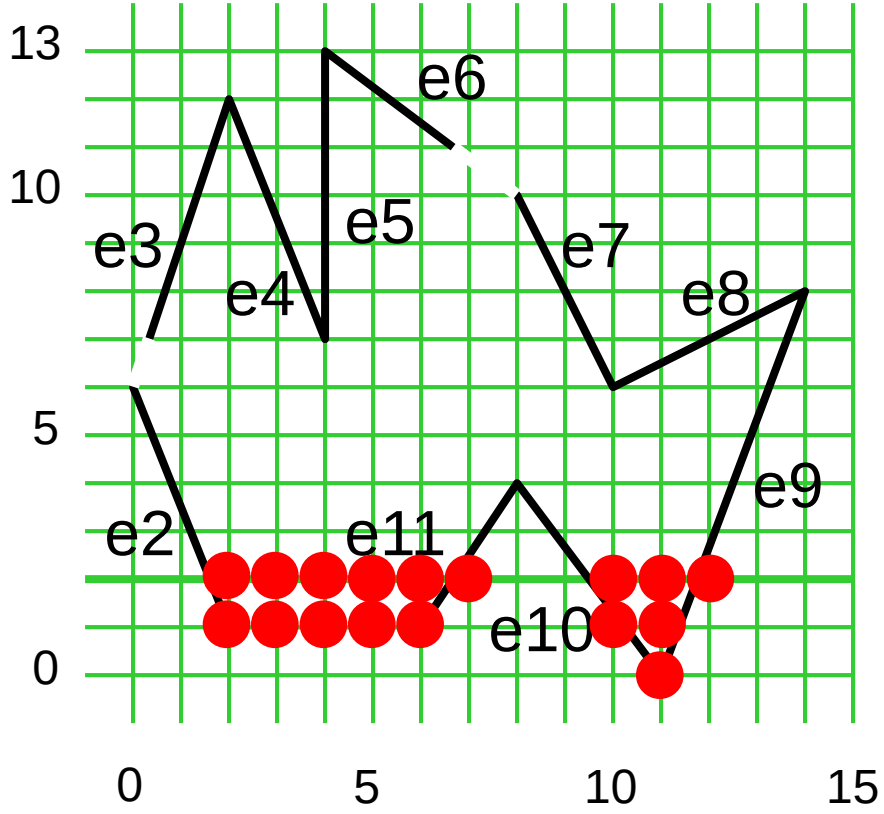
SCAN LINE ALGORITHM



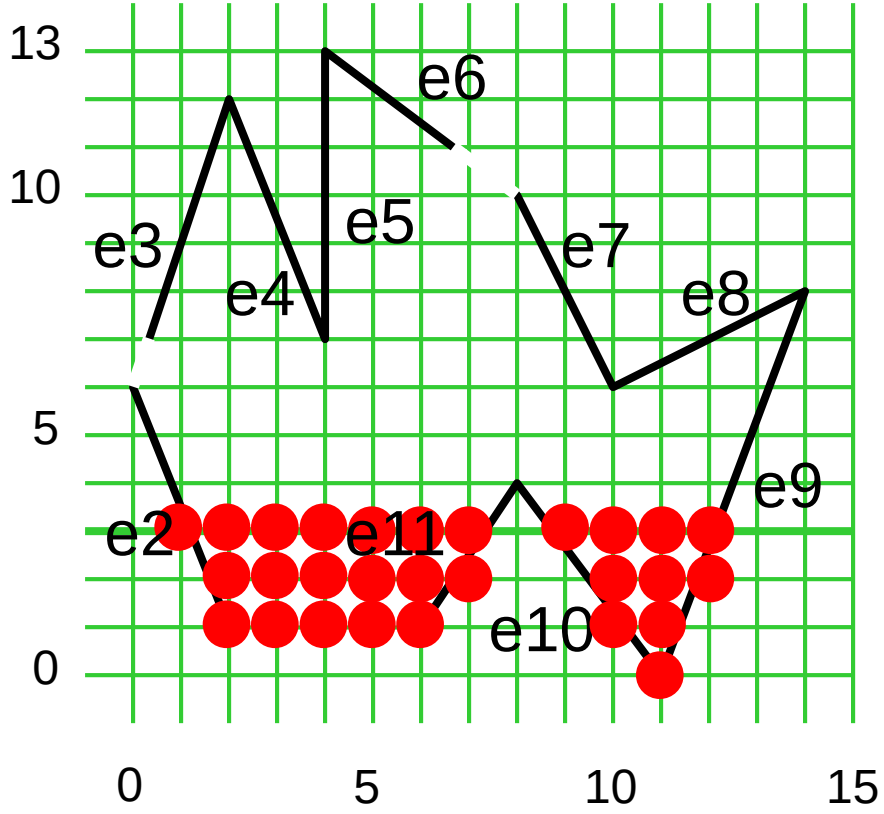
Running the Algorithm



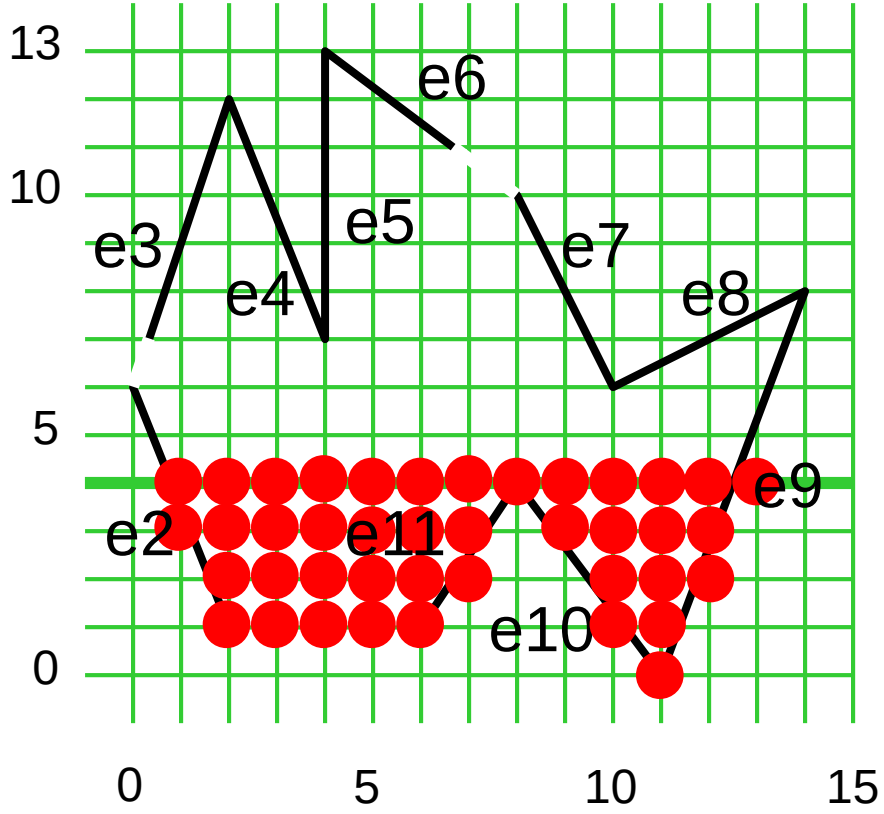
Running the Algorithm



Running the Algorithm

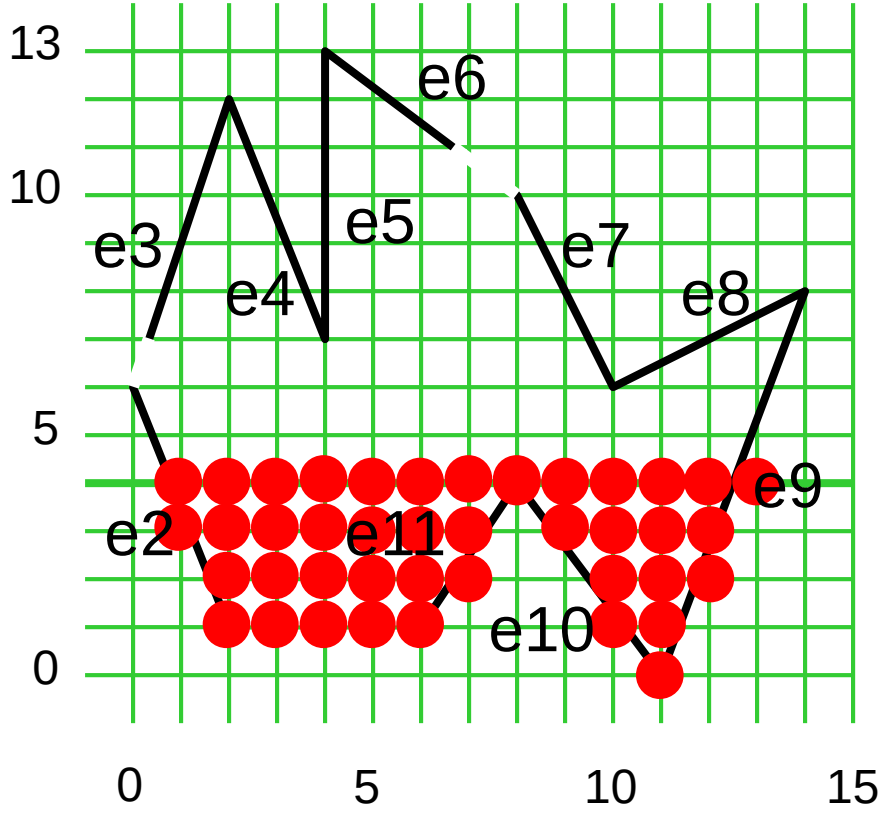


Running the Algorithm

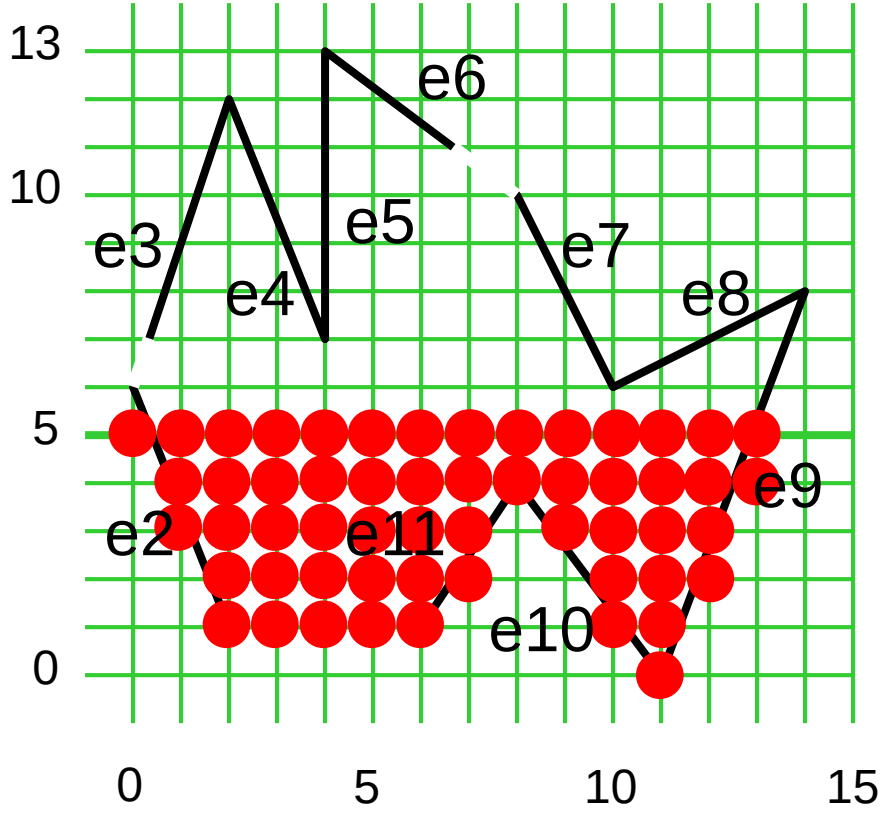


Remove these edges.

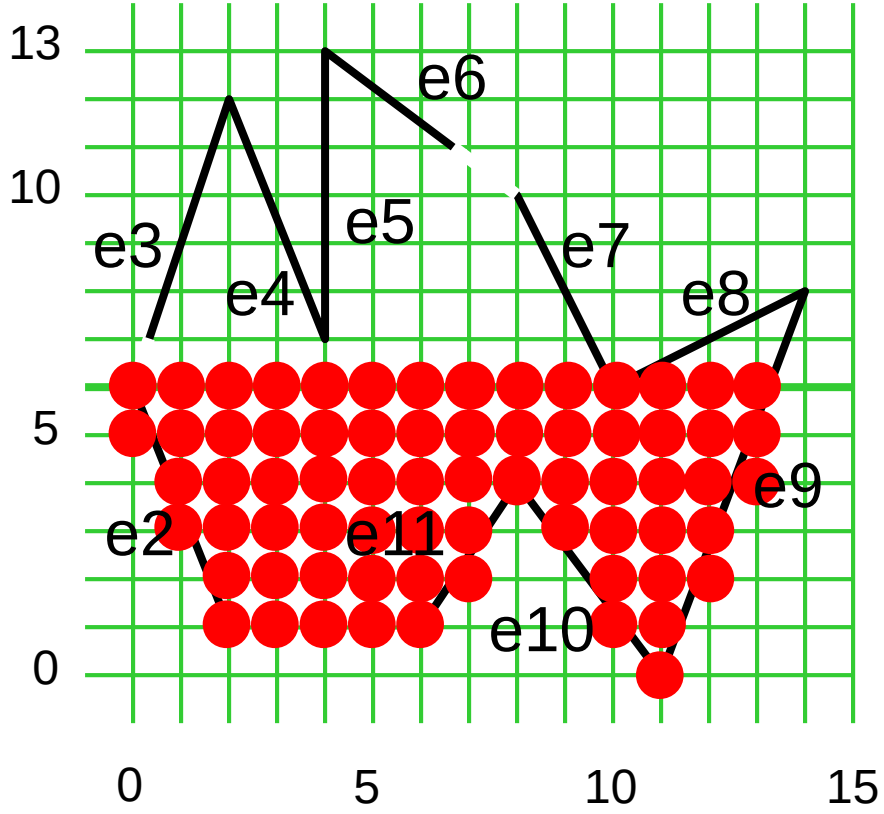
Running the Algorithm



Running the Algorithm



Running the Algorithm

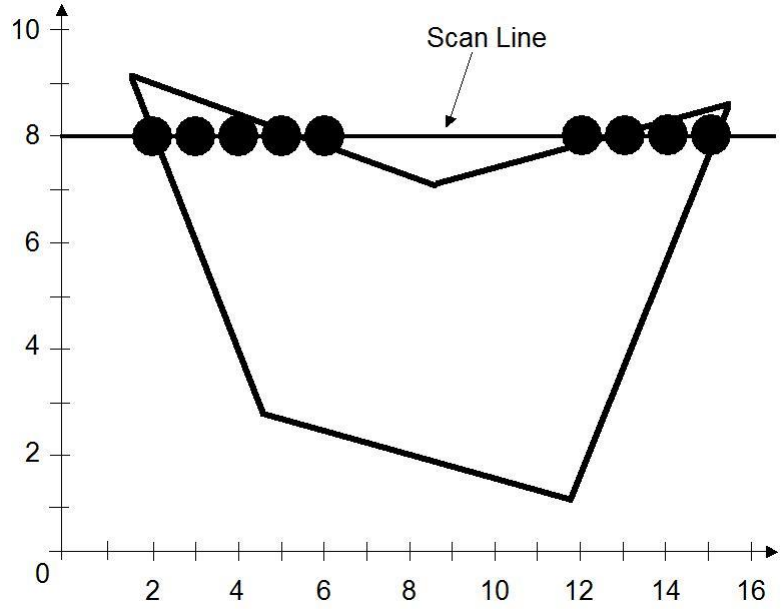


Running the Algorithm

The Scan-Line Polygon Fill Algorithm

The scan-line polygon-filling algorithm involves

- the **horizontal scanning** of the polygon from its **lowermost** to its **topmost** vertex,
- identifying which edges intersect the scan-line,
- and finally drawing the interior horizontal lines with the specified fill color. process.



Example

- Consider the following polygon:

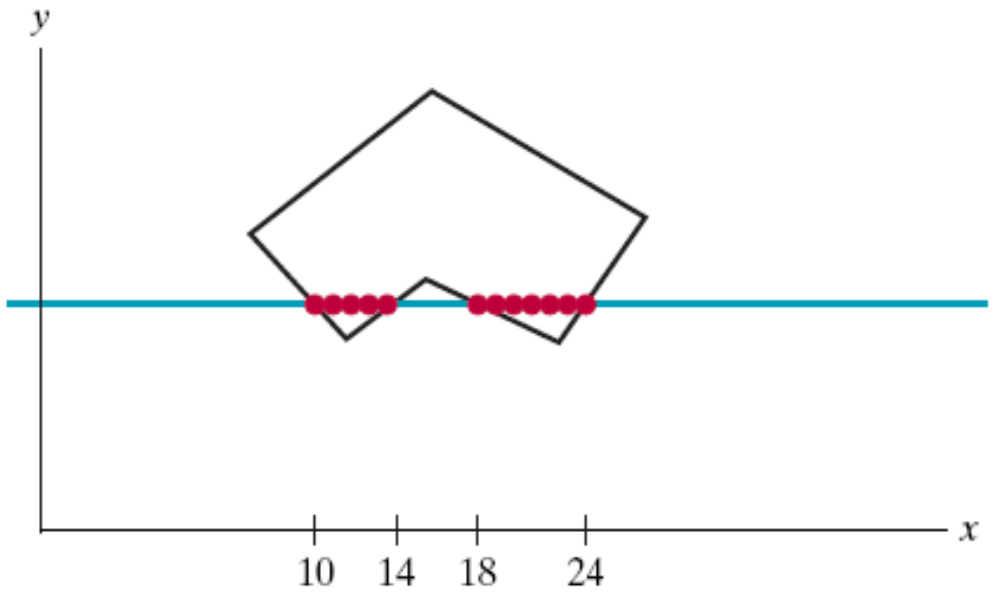


FIGURE 4–20 Interior pixels along a scan line passing through a polygon fill area.

Example

- For each scan line that crosses the polygon, the edge intersections are sorted from left to right, and then the pixel positions between, and including, each intersection pair are set to the specified fill color.

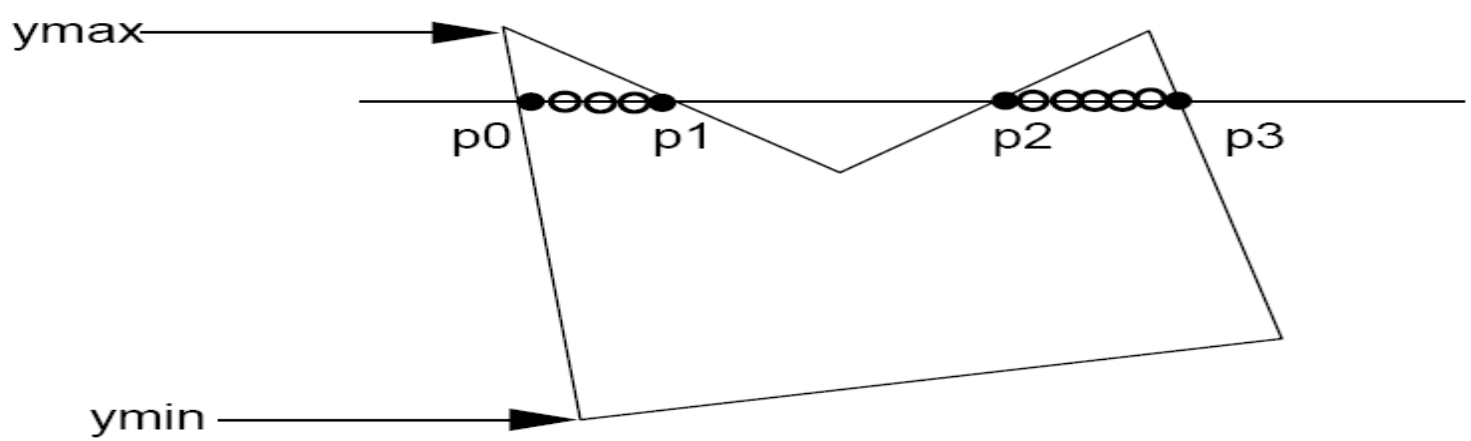
- In the previous Figure, the four pixel intersection positions with the polygon boundaries define two stretches of interior pixels.

Scanline Fill Algorithm

- Intersect scanline with polygon edges
- Fill between pairs of intersections
- Basic algorithm:

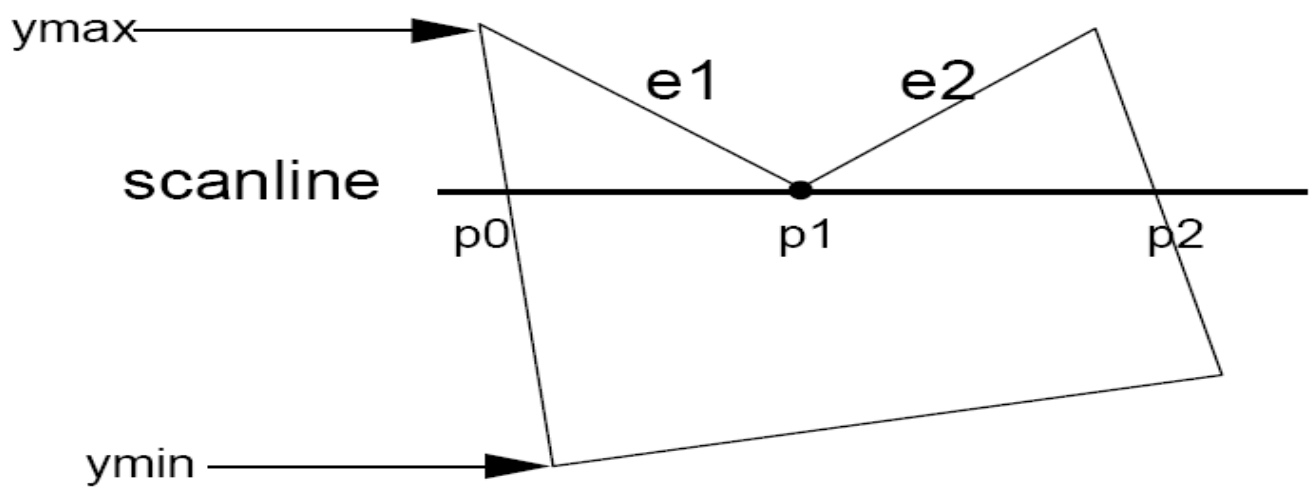
  For y = ymin to ymax

  1) intersect scanline y with each edge
  2) sort interesections by increasing x
     [p0,p1,p2,p3]
  3) fill pairwise (p0 –> p1, p2–> p3, ….)



Special handling (cont'd)

c) Intersection is an edge end point



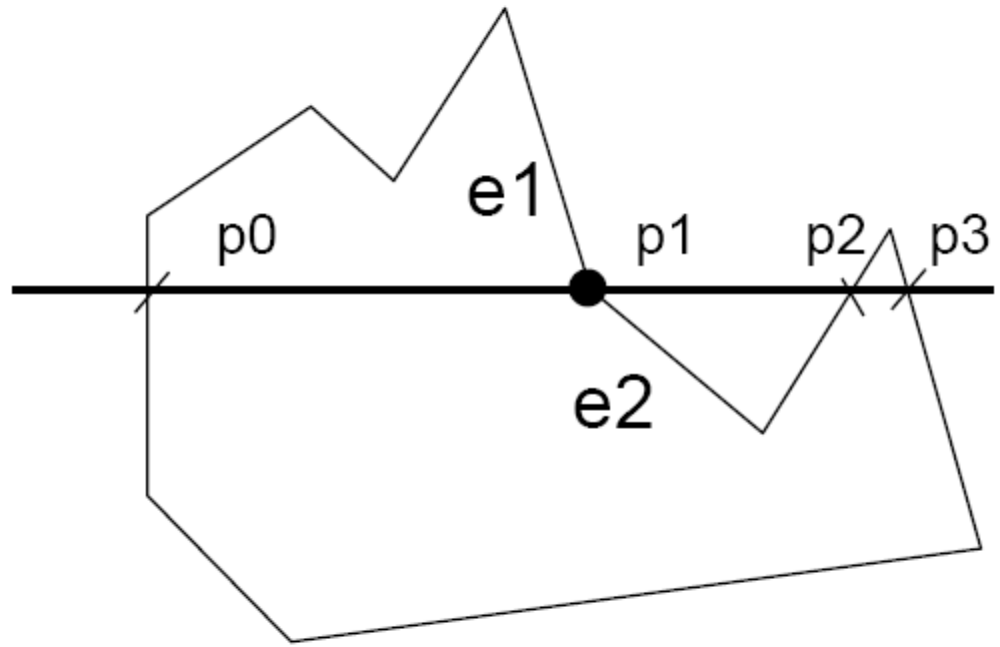
Intersection points: (p0, p1, p2) ???

–> (p0,p1,p1,p2) so we can still fill pairwise

–> In fact, if we compute the intersection of the scanline with edge e1 and e2 separately, we will get the intersection point p1 twice. Keep both of the p1.

Special handling (cont'd)

c) Intersection is an edge end point (cont'd)



However, in this case we don't want to count p1 twice (p0,p1,p1,p2,p3), otherwise we will fill pixels between p1 and p2, which is wrong

Polygon Fill Algorithm

- Consider the next Figure.
- It shows two scan lines that cross a polygon fill area and intersect a vertex.
- Scan line *y*’ intersects an even number of edges, and the two pairs of intersection points along this scan line correctly identify the interior pixel spans.
- But scan line *y* intersects five polygon edges.

Polygon Fill Algorithm

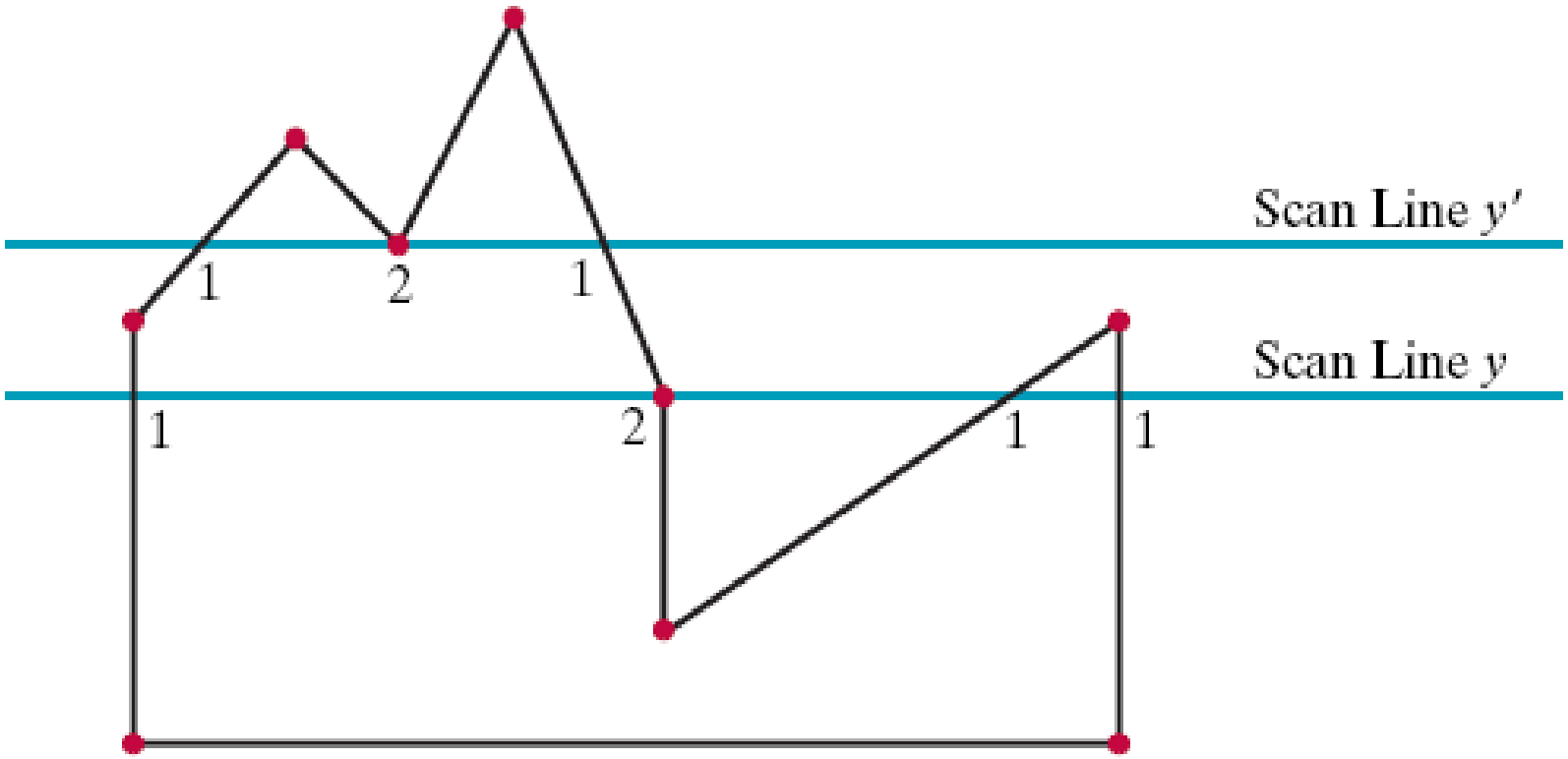


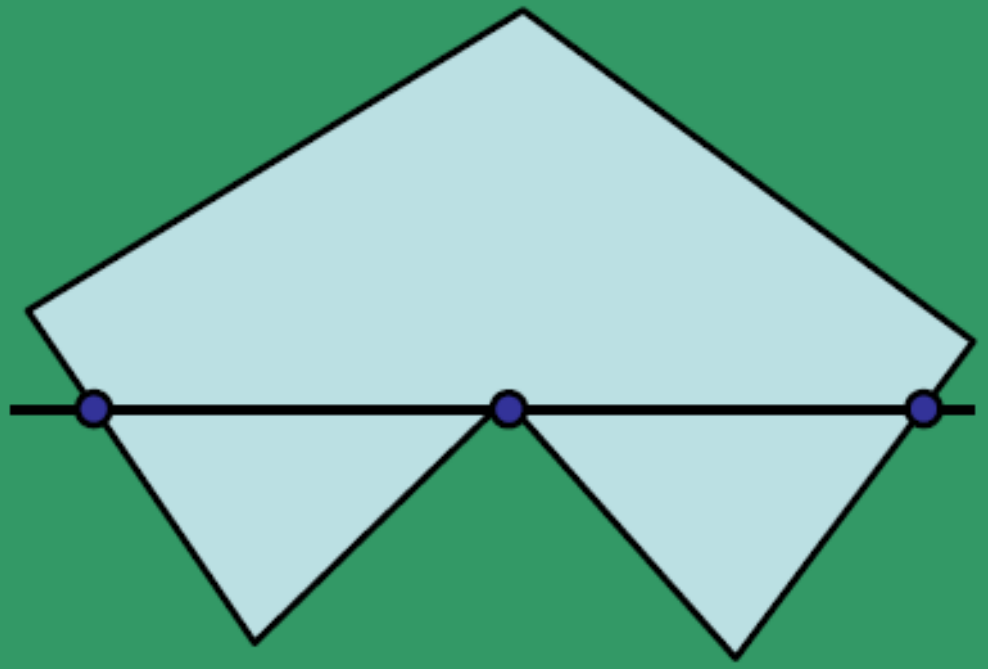
FIGURE 4–21 Intersection points along scan lines that intersect polygon vertices. Scan line y generates an odd number of intersections, but scan line y' generates an even number of intersections that can be paired to identify correctly the interior pixel spans.

Polygon Fill Algorithm

- To identify the interior pixels for scan line *y*, we must count the vertex intersection as only one point.

- Thus, as we process scan lines, we need to distinguish between these two cases.

Two different cases of scanlines passing through the vertex of a polygon



Case - I

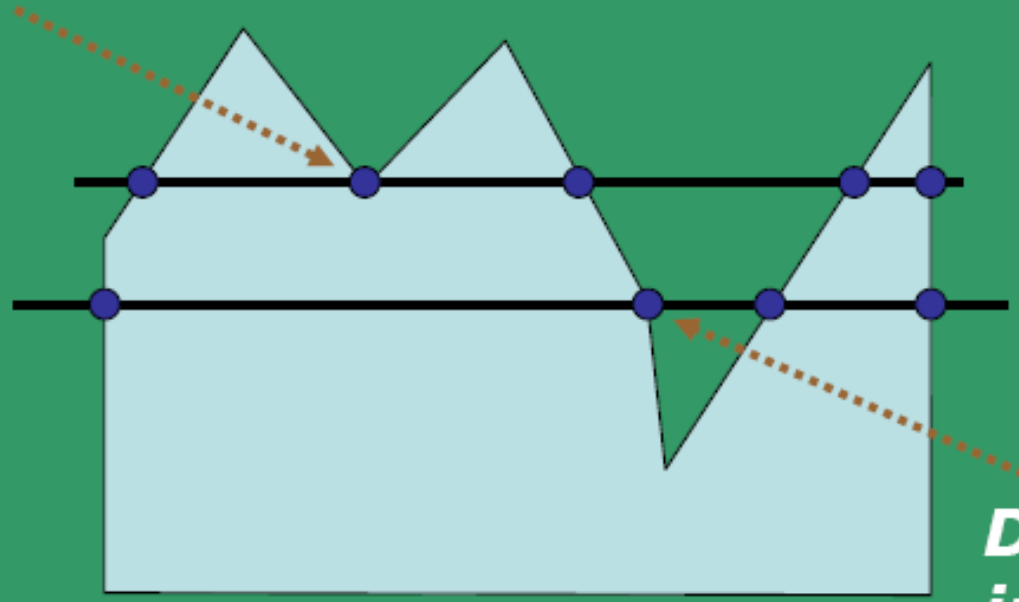
← *Add one more intersection:*

3 -> 4

Case - II

Add one more intersection:

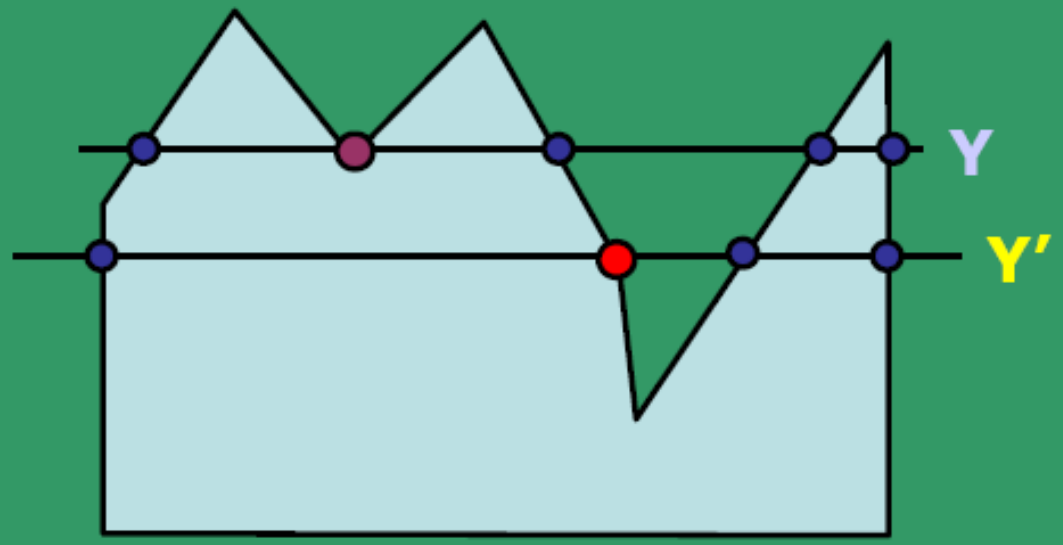
5 -> 6;



Do not add intersection, keep 4;

HOW ??

What is the difference between the intersection of the scanlines Y and Y', with the vertices?

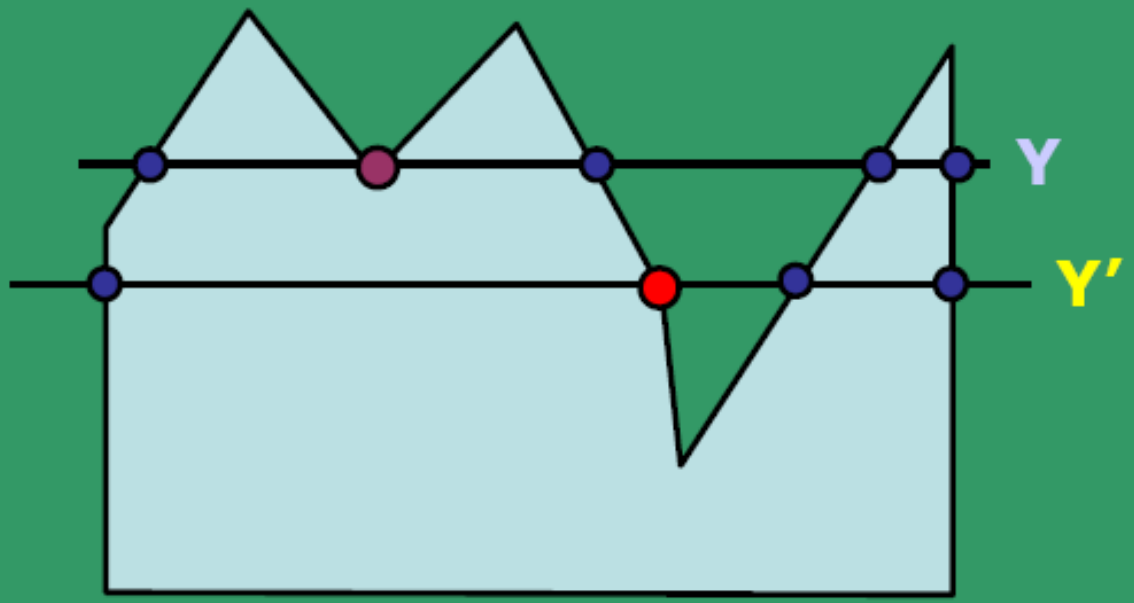


For Y, the edges at the vertex are on the same side of the scanline.

Whereas for Y', the edges are on either/both sides of the vertex.

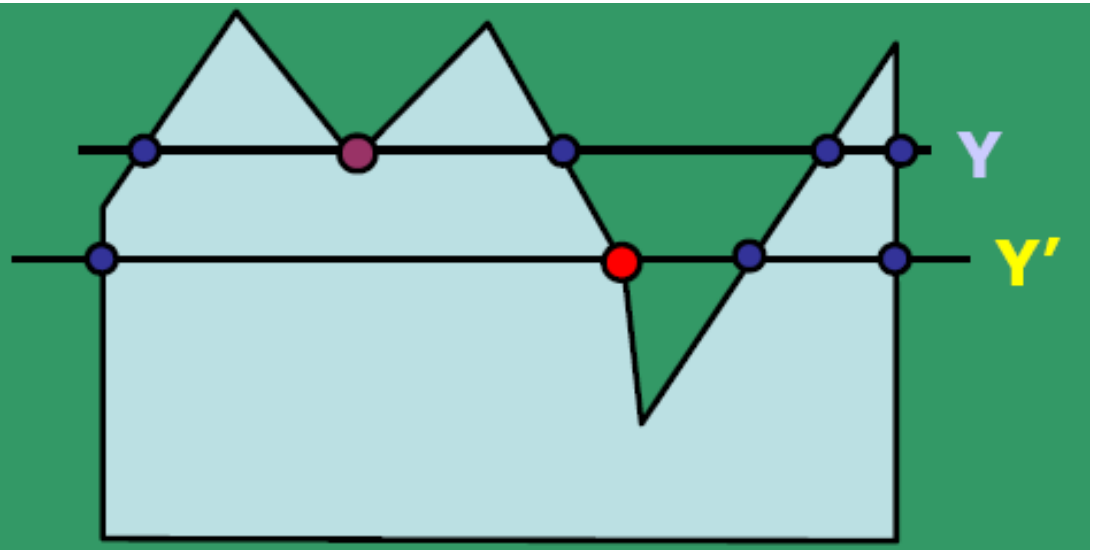
In this case, we require additional processing.

Vertex counting in a scanline



- **Traverse** along the polygon boundary clockwise (or counter- clockwise) and
- **Observe** the *relative change in Y-value* of the edges on either side of the vertex (i.e. as we move from one edge to another).

Vertex counting in a scanline



Check the condition:

If *end-point Y values* of two consecutive edges *monotonically increase or decrease*, count the middle vertex as a single intersection point for the scanline passing through it.

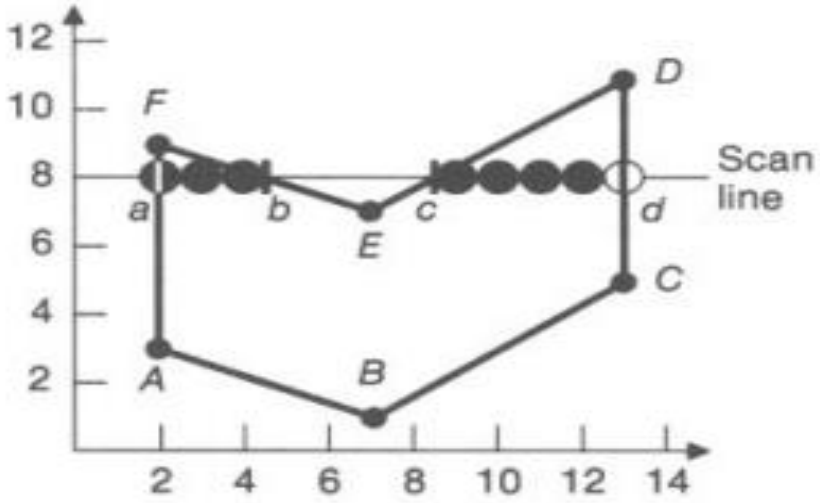
Else the shared vertex represents a *local maximum (or minimum)* on the polygon boundary. *Increment the intersection count*.

Scan Line Algorithm

Start with max y and move to min y or vice versa

For each scan line:

- Find the intersections of the scan line with all edges of the polygon.
- Sort the intersections by increasing x coordinate.
- Fill in all pixels between pairs of intersections
- For scan line number 8 the sorted list of x-coordinates is (2, 4, 9, 13) Therefore draw line b/w (2,4) & (9,13)



Assume that we are taking scan line from min y and moving towards max y

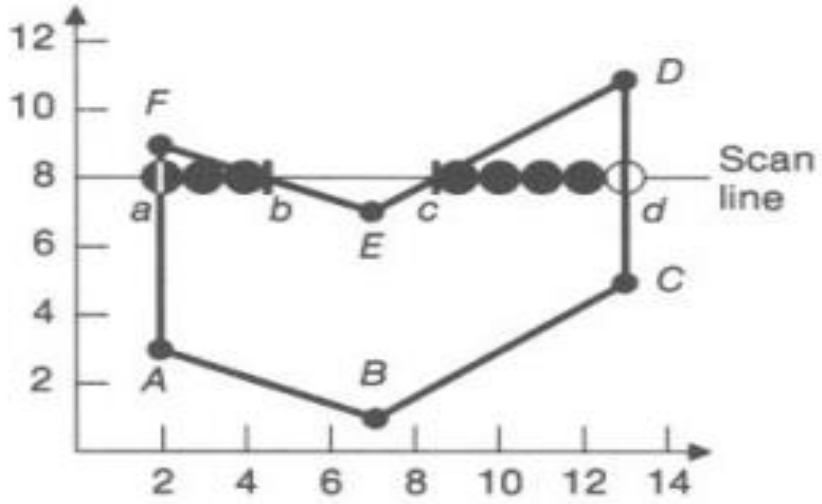
- We know that for every scan line we have to calculate x intersection with every polygon side. We can determine the next *x* intersection value as

- $x_{i+1}=x_i + 1/m$ where m is the slope of edge

- It is very easy to identify which polygon sides should be tested for x-intersection, if we sort edges in order of min *y* and find active edge list (Active edge list for scan line contains all the edges crossed by that scan line).

- Two pointers are used to indicate the beginning and ending of active edge list. As the scan line move up the picture the beginning and end pointers also moved up to eliminate the edges which end above the current scan line and include the new edges which now start on or above the current scan line .

Edge List and Active Edge List



Parametric Polynomial Curves

- We'll use polynomial parametric curves, where the functions are all **polynomials** in the parameter.

$$P_t = \sum_{i=0}^{n} A_i t^i$$

- $P_t = A_0 + A_1 t + A_2 t^2 + A_3 t^3 \ \ldots$
- Advantages
  - easy (and efficient) to compute
  - infinitely differentiable

- We'll also assume that t varies from 0 to 1

Bezier Curves

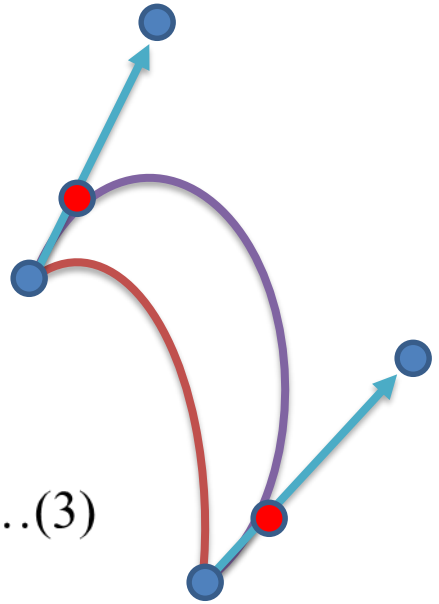
- $P_t = T \,.\, M \,.\, G$ ... (1)
- For the case of Bezier,
- $P_t = T \,.\, M_B \,.\, G_B$ ... (2)
- Where $G_B = \begin{vmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{vmatrix}$, *P0 .. P3* are 4 control points
- We know that $G_H = \begin{vmatrix} P_o \\ P_3 \\ G_0 \\ G_3 \end{vmatrix} = \begin{vmatrix} P_o \\ P_3 \\ P_1 - P_0 \\ P_3 - P_2 \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{vmatrix} \cdot \begin{vmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{vmatrix}$

Bezier Curves

- Relationship between G_H and G_B

- $$G_H = \begin{vmatrix} P_o \\ P_3 \\ G_0 \\ G_3 \end{vmatrix} = \begin{vmatrix} P_o \\ P_3 \\ P_1 - P_0 \\ P_3 - P_2 \end{vmatrix}$$

- $$G_{HB} = \begin{vmatrix} P_o \\ P_3 \\ 3G_0 \\ 3G_3 \end{vmatrix} = \begin{vmatrix} P_o \\ P_3 \\ 3(P_1 - P_0) \\ 3(P_3 - P_2) \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{vmatrix} \cdot \begin{vmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{vmatrix} \quad \ldots(3)$$



Bezier Curves

- 4 point based Hermite curve is,
- $$P_t = T \,.\, M_H \,.\, G_H = T \,.\, M_H \,. \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{vmatrix} \cdot \begin{vmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{vmatrix}$$
- And the Bezier curve in Hermite form,
- $$P_t = T \,.\, M_H \,.\, G_{HB} = T \,.\, M_H \,. \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{vmatrix} \cdot \begin{vmatrix} P_o \\ P_1 \\ P_2 \\ P_3 \end{vmatrix}$$, where $G_{BH} = M_{HB} \,.\, G_B$
- $$P_t = T \,.\, M_H \,.\, G_{HB} = T \,.\, M_H \,. \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{vmatrix} . G_B = T \,.\, M_B \,.\, G_B \quad \ldots (4)$$

Bezier Curves

- Therefore M_B can be written as,
- $$M_B = M_H \cdot \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{vmatrix} = \begin{vmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{vmatrix} \cdot \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{vmatrix}$$
- $$M_B = \begin{vmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{vmatrix}$$

Finding *P(t)* in Bezier Curve

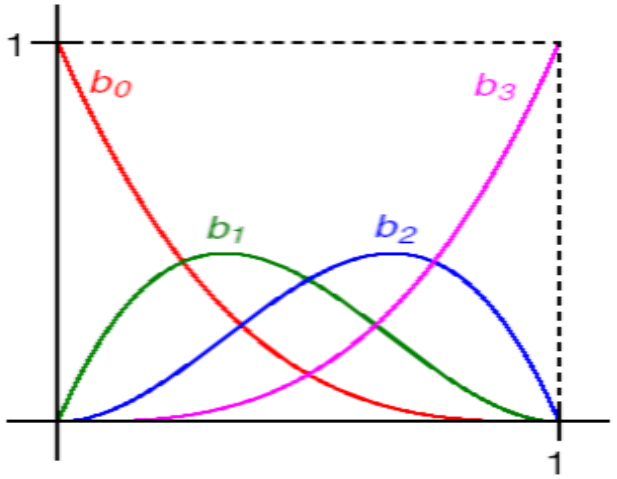
- Bernstein polynomials
- In general
- $P_{(t)} = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} P_i$

  where "n choose i" is $\binom{n}{i} = \frac{n!}{(n-i)!\,i!}$

- This defines **Bezier curves**
- What is the relationship between the number of control points and the degree of the polynomials?

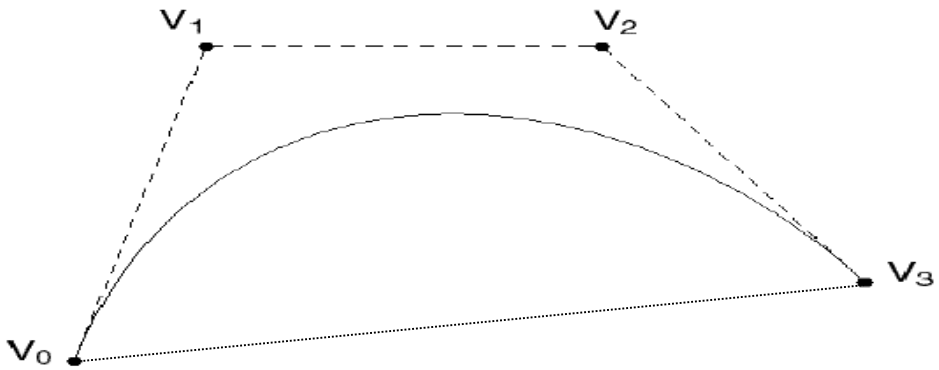
Finding $P(t)$ in Bezier Curve

- The coefficients of the control points are a set of functions called the **Bernstein polynomials** also the blending functions.
- For degree 3, we have:
- $(1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)p_2 + t^3P_3$
- $B_{B0} = (1 - t)^3$
- $B_{B1} = 3(1-t)^2t$
- $B_{B2} = 3(1-t)\,t^2$
- $B_{B3} = t^3$



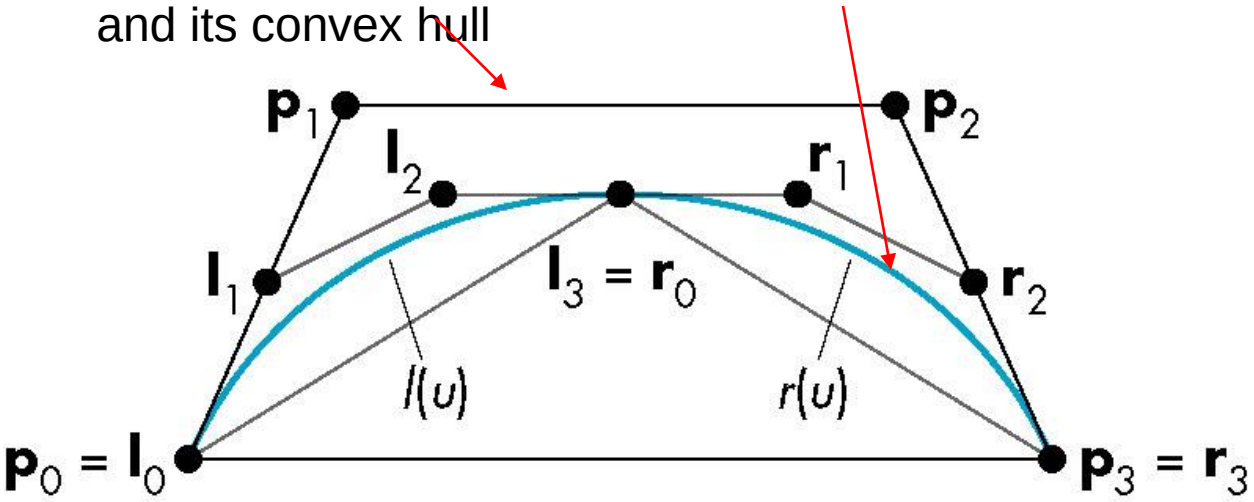
Useful properties of Bezier curve

- Bernstein polynomials has some useful properties in [0,1]:
  - each Bernstein coefficient is positive
  - sum of all four coefficients is always exactly 1
- These properties together imply that the curve lies within the **convex hull** of its control points. (convex hull is the smallest convex polygon that contains the control points)



Splitting a Cubic Bezier

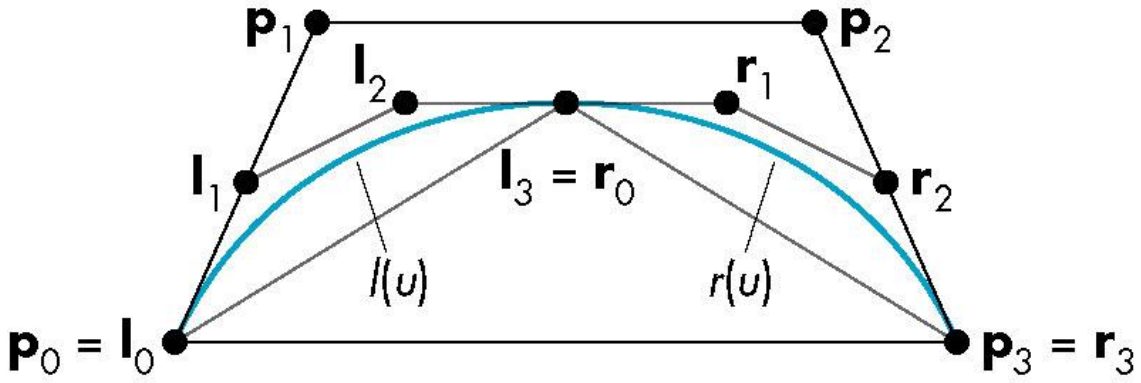
p_0, p_1 , p_2 , p_3 determine a cubic Bezier polynomial and its convex hull



Consider left half l(u) and right half r(u)

l(t) and r(t)

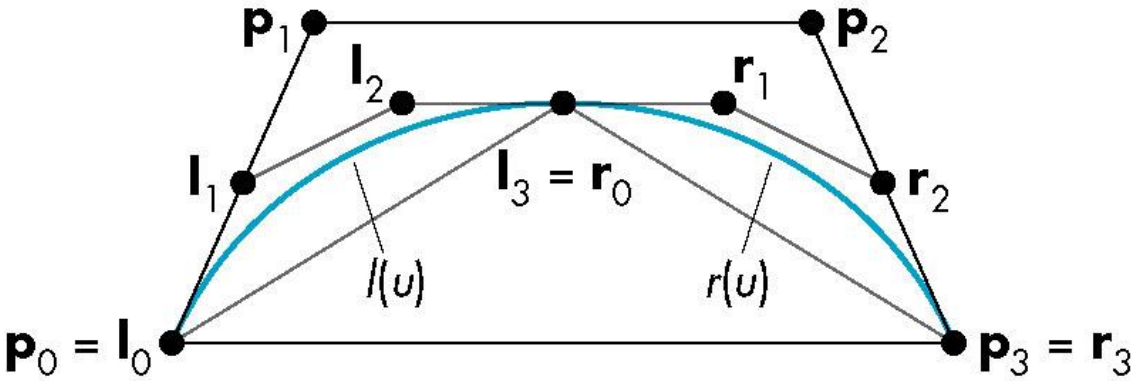
Since l(t) and r(t) are Bezier curves, we should be able to find two sets of control points $\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$ that determine them



Convex Hulls

$\{l_0, l_1, l_2, l_3\}$ and $\{r_0, r_1, r_2, r_3\}$each have a convex hull that that is closer to $p(t)$ than the convex hull of $\{p_0, p_1, p_2, p_3\}$ This is known as the *variation diminishing property*.

The polyline from l_0 to l_3 (= r_0) to r_3 is an approximation to $p(t)$. Repeating recursively we get better approximations.



Efficient Form

Assuming t = 0.5

$l_0 = p_0$

$l_1 = ½(p_0 + p_1)$

$l_2 = ½(l_1 + ½(p_1 + p_2))$

$= ¼(p_0 + 2p_1 + p_2)$

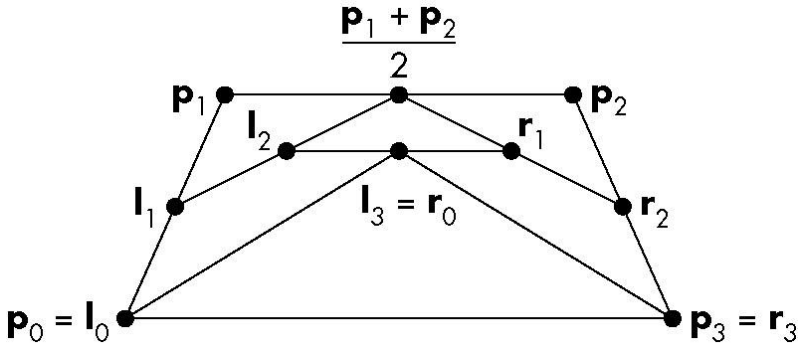
$r_3 = p_3$

$r_2 = ½(p_2 + p_3)$

$r_1 = ¼(p_1 + 2p_2 + p_3)$

$l_3 = r_0 = ½(l_2 + r_1)$

$= ⅛(p_0 + 3p_1 + 3p_2 + p_3)$



Requires only shifts and adds!

Left and Right Segments

The geometric constrain of the left segment assuming t=0.5 can be written as

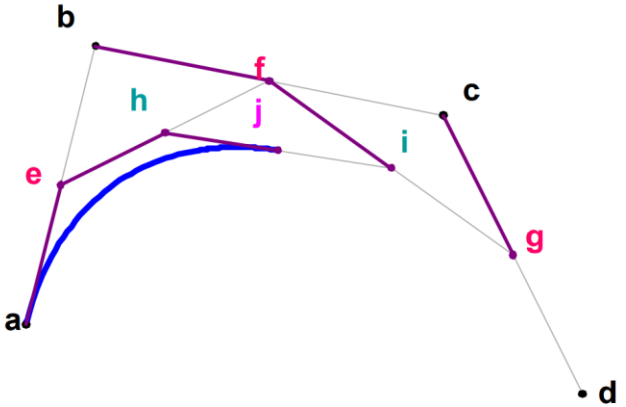
$$G_{BL} = 1/8 \begin{vmatrix} 8 & 0 & 0 & 0 \\ 4 & 4 & 0 & 0 \\ 2 & 4 & 2 & 0 \\ 1 & 3 & 3 & 1 \end{vmatrix} \bullet \begin{vmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{vmatrix}$$

And the right segment is

$$G_{BR} = 1/8 \begin{vmatrix} 1 & 3 & 3 & 1 \\ 0 & 2 & 4 & 2 \\ 0 & 0 & 4 & 4 \\ 0 & 0 & 0 & 8 \end{vmatrix} \bullet \begin{vmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{vmatrix}$$

Geometric Construction by De Casteljau

- Casteljau's algorithm provides a method for geometrically constructing the Bezier curve. In the following example construction of a cubic Bezier is demonstrated.



- For the case of a cubic Bezier, we consider the three limbs of the open control polygon *ab*, *bc*, and *cd*. Next create the intermediate points *e*, *f* and *g* in the ratios $ae/ab = bf/bc = cg/cd = t$ (given value of the parameter). Continuing iteratively we obtain the point j on the curve. Similarly a series of values of *t* give rise to the corresponding ratios and hence the points on the Bezier curve.