Graphics 2008/2009, period 1

Lecture 8

Triangle rasterization
and
surface shading

Overview

Today, we will finish "the basic stuff" (chapter 1-9 of the book):

- We will *not* look into signal processing (chapter 4)
- We will look at triangle rasterization (cf. chapter 3, but differently covered here)
- Also at shading (cf. chapter 9, also some differences here)
- And again at Z-buffering (not covered in the book)

scanline

After the midterm exam, we will cover some "advanced topics":

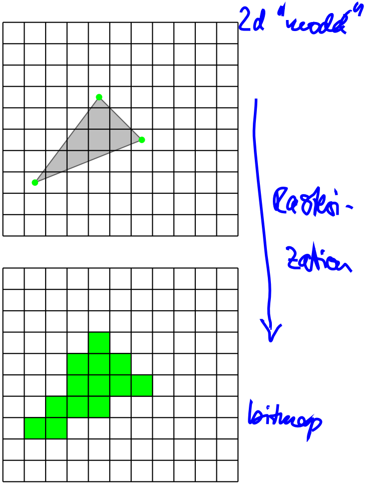
- Ray tracing (chapter 10)
- Texture mapping (chapter 11)
- Graphics pipeline, part II (chapter 12)
- Radiosity and shadows (not in the book)

Rasterizing triangles

We know how to project the vertices of a triangle in our model onto pixel centers.

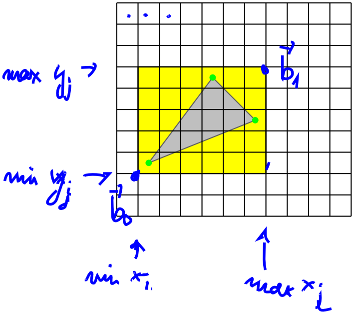
To draw the complete triangle, we have to decide which pixels to turn on.

For now, let's assume that all pixels of the triangle get the same color.



Limiting the number pixels to consider

Instead of testing for all pixels in the image if they belong to the triangle, our book suggests to test only pixels in the bounding box of the triangle.

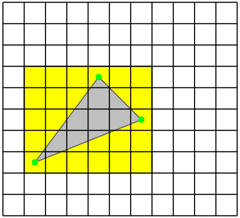


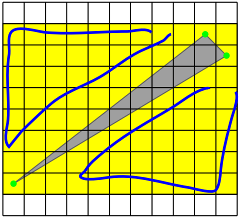
$$\vec{b}_0 = (\min\{x_i\}, \min\{y_j\})$$

$$\vec{b}_1 = (\max\{x_i\}, \overset{\max}{\cancel{\min}}\{y_j\})$$

Limiting the number pixels to consider

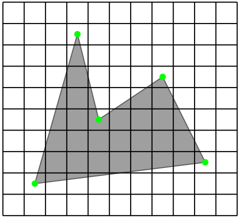
However, for long and skinny triangles that are diagnonally oriented, considering all pixels in their bounding box may still be quite inefficient.





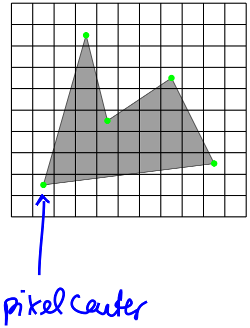
Rasterizing general polygons

Let's look at a more efficient way to draw triangles. The method actually works for general polygons; for triangles, it becomes even more efficient.



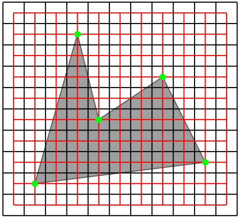
From pixel centers to pixel coordinates

First, let's move our attention from pixels to the underlying coordinate system where the pixel centers have integer coordinates.



From pixel centers to pixel coordinates

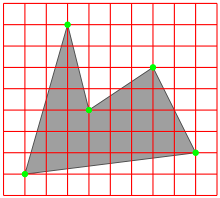
First, let's move our attention from pixels to the underlying coordinate system where the pixel centers have integer coordinates.



From pixel centers to pixel coordinates

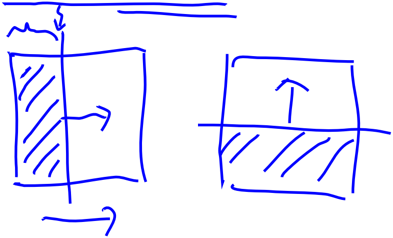
First, let's move our attention from pixels to the underlying coordinate system where the pixel centers have integer coordinates.

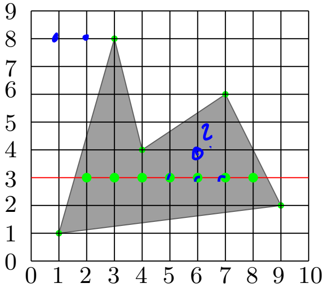
$\frac{1}{2}$



Scanline conversion

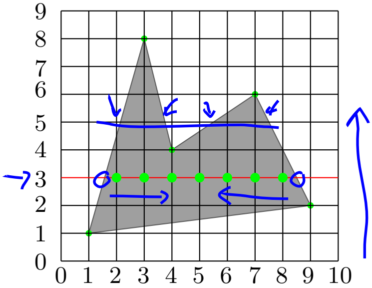
Idea: instead of looking at each pixel individually, we draw our polygon one scanline at a time, from bottom to top, thus taking advantage of scanline coherence.





Scanline conversion

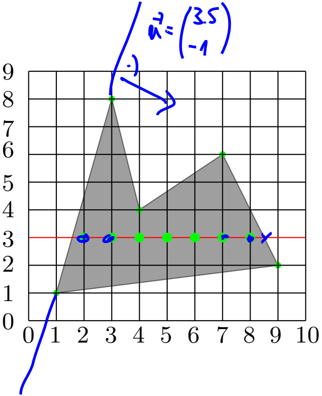
Scanline 3 intersects two edges; these are called active edges. If we know the intersections of the active edges with the current scanline, then we know which pixels to set.



Computing intersections

The left active edge runs from $(1, 1)$ to $(3, 8)$. The implicit equation for the line through these points is $3.5x - y - 2.5 = 0$.

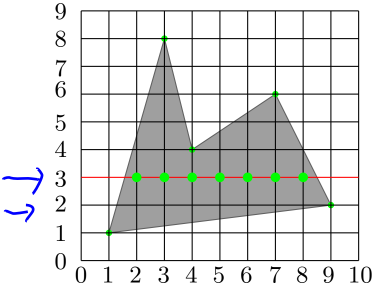
The x-coordinate of the intersection of the edge with scanline 3 is $1\frac{4}{7}$, so we know that we have to set pixels starting from $x = 2$.



Computing intersections incrementally

Computing intersections of scanlines and edges requires a division. These are expensive, and we'd like to avoid them.

We use the fact that the intersection of an edge with scanline i is related to the intersection with scanline $i-1$. This is called vertical coherence.



Computing intersections incrementally

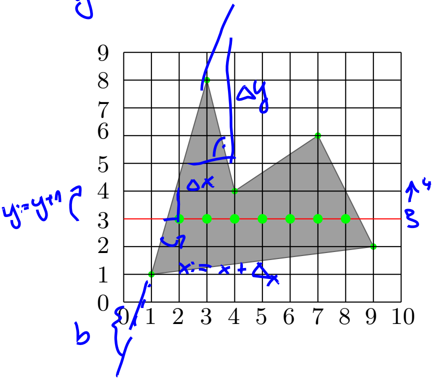
slope - intercept repr.

$y = ax + b$

For the left active edge, the increase in x-coordinate from one scanline to the next is

→ $\Delta_x = \frac{3-1}{8-1} = \frac{2}{7}$. $\Delta y \stackrel{!}{=} 1 \stackrel{!}{=} \Delta_y$

$$\text{slope} = \frac{\Delta y}{\Delta x} = \frac{8-1}{3-1} = \frac{1}{\Delta_x}$$



Computing intersections incrementally

Example: The x-coordinate at scanline 1 is obviously 1.

So we have:

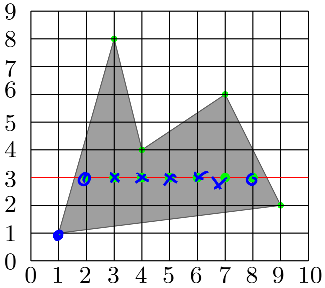
scanline	x-coordinate
1	1
2	$1\frac{2}{7}$
3	$1\frac{4}{7}$
4	$1\frac{6}{7}$

y-coord

+1

Δx

Δx

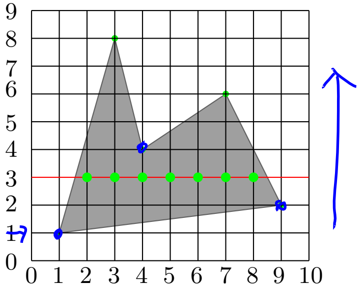


Data structures: edge table (ET)

In the Edge Table (ET), we maintain for every scanline a list of edges that start at the scanline.

Each edge record contains:

- The x-coordinate of the lowest vertex.
- The y-coordinate of the highest vertex
- The value of Δ_x for the edge.



ET:

$1\colon\ (1, 8, 2/7), \quad (1, 2, 8)$

$2\colon (9, 6, -1/2)$

$4\colon (4, 8, -1/4), \quad (4, 6, 1.5)$

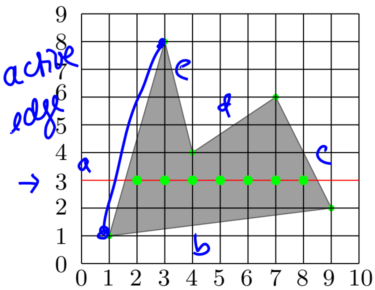
Data structures: active edge table (AET)

When we proceed from one scanline to the next, we also maintain an Active Edge Table (AET). This table contains the edge record from the ET that are intersected by the current scanline.

initially(!)

Scanline #3:

active edges : a, c



ET:
1: $(1, 8, 2/7)$, $(1, 2, 8)$
2: $(9, 6, -1/2)$
4: $(4, 8, -1/4)$, $(4, 6, 1.5)$

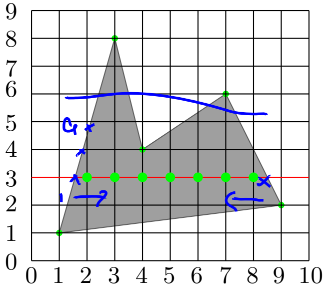
Data structures: active edge table (AET)

Whenever we move to the next scanline, we add the Δ_x-values of the active edges to the x-value.

~~Scanl. # 3~~

Scanl. # 3		a	c	
0	:	$\cdot/.$	,	
1	:	$(1, 8, \frac{2}{7})$,	,	
2	:	$(1\frac{2}{7}, 8, \frac{2}{7})$	$(9, 6, -\frac{1}{2})$	+1
3	:	$(1\frac{4}{7}, 8, \frac{2}{7})$	$(8\frac{1}{2}, 6, -\frac{1}{2})$	(y)

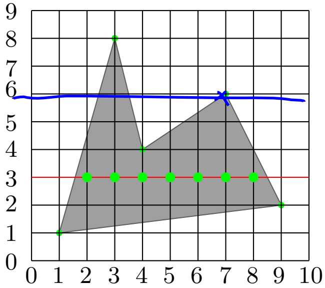
Δ_x (x)



ET:
1: $(1, 8, 2/7)$, $(1, 2, 8)$
2: $(9, 6, -1/2)$
4: $(4, 8, -1/4)$, $(4, 6, 1.5)$

Active edge table

An edge record is removed from the AET when the current scanline reaches the top of the edge. Recall that the 2nd entry of the edge record stores the y-coordinate of the top vertex.



ET:

1: $(1, 8, 2/7)$, $(1, 2, 8)$

2: $(9, 6, -1/2)$

4: $(4, 8, -1/4)$, $(4, 6, 1.5)$

Scanline conversion

So the whole algorithm becomes:

AET $= \emptyset$	
for $i = 0$ to $n-1$ do	for each scanline
update(AET,i)	delete edge records for which $y == i$
	add Δ_x to x-values
append(AET,ET$[i]$)	add edges starting here
sort(AET)	
JoinLines(AET,i)	set pixels between pairs of edges

Active edge table

0: /

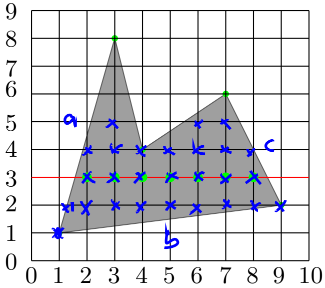
1: $(1, 8, 2/7)$, $(1, 2, 8)$ → draw 1 pixel

2: $(1\frac{2}{7}, 8, \frac{2}{7})$, $(9, 6, -\frac{1}{2})$

3: $(1\frac{4}{7}, 8, \frac{2}{7})$, $(8.5, 6, -\frac{1}{2})$

4: $(1\frac{6}{7}, 8, \frac{2}{7})$, $(8, 6, -\frac{1}{2})$, $(4, 8, -\frac{1}{4})$, $(4, 6, 1.5)$

5: ...



ET:

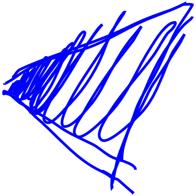
1: $(1, 8, 2/7)$, $(1, 2, 8)$

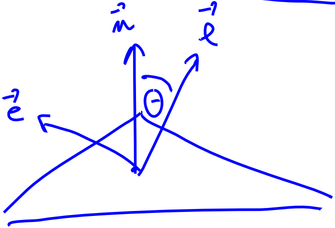
2: $(9, 6, -1/2)$

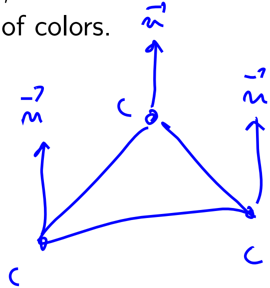
4: $(4, 8, -1/4)$, $(4, 6, 1.5)$

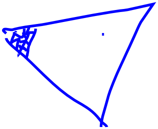
Linear interpolation

So far, we have set pixels to a fixed color. However, if we compute diffuse reflection at the vertices of the triangle, we can extend the scanline algorithm to do bilinear interpolation of colors.









$$\underline{c} = c_r \cdot c_l \cdot \underbrace{\vec{n} \cdot \vec{l}}_{\sim \cos\theta}$$

reflection fac.   RGB int. [illegible]

Linear interpolation

So far, we have set pixels to a fixed color. However, if we compute diffuse reflection at the vertices of the triangle, we can extend the scanline algorithm to do bilinear interpolation of colors.

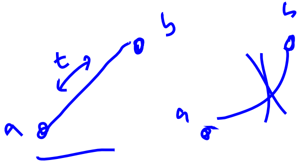
Def.: Given 2 values or vectors a, b:

lin. interpol.: $p = (1-t)\cdot \underline{a} + t \cdot \underline{b}$

$t = 0 \rightsquigarrow a$

$t = 1 \rightsquigarrow b$

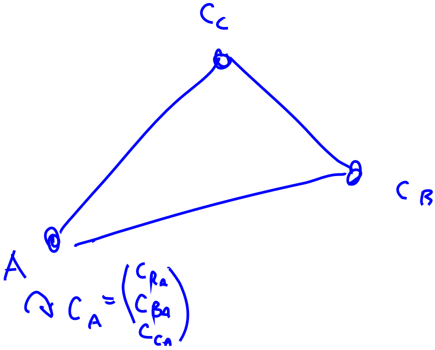
$0 < t < 1 \rightsquigarrow$ ..



Gouraud shading

For every scanline, color-values at the edges of the triangle are computed by linearly interpolating the color values at the vertices.

This can be done incrementaly, by precomputing Δ_r, Δ_g, and Δ_b values, and adding these to the r, g and b values of the previous scanline.

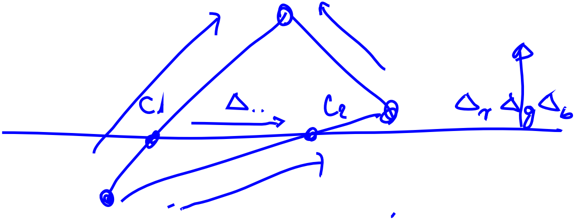


Gouraud shading

Colors along each scanline are also determined by linear interpolation. Again, this is done incrementally, by precomputing Δ values.

So for each triangle, we do four devisions for each of its edges, to compute the Δ_x, Δ_r, Δ_g, and Δ_b values, and three divisions for every scanline it spans, to compute the color increments in horizontal direction.

Gouraud interpol.



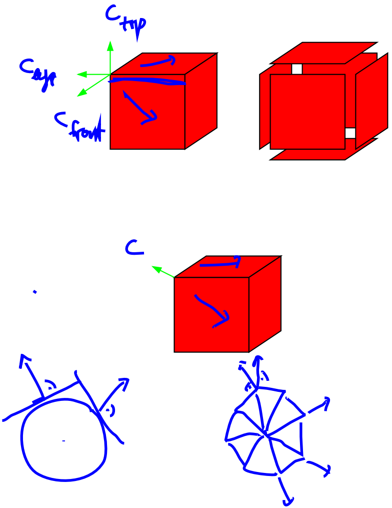
Gouraud shading

But how do we compute diffuse lighting at the vertices of the triangles?

It depends.

We may specify that normals are indeed perpendicular to the triangles.

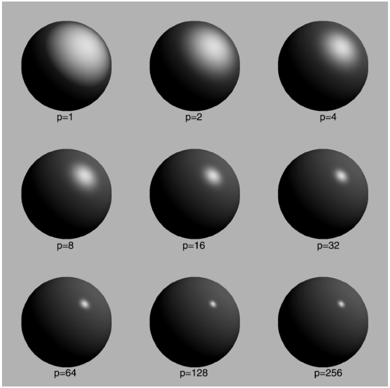
We may also make triangles have their vertices shared, and somehow interpolate vertex normals.



Recap: Glossy reflection / Phong shading

Remember: to model objects with some highlights or hotspots, we use phong reflection.

Phong exponent p



(image source: textbook, p. shirley, fig. 9.6, page 195)

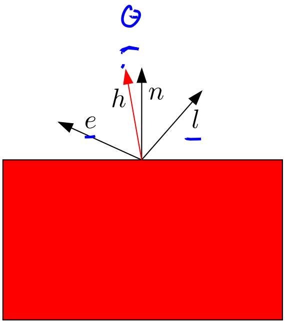
Recap: Glossy reflection / Phong shading

Reflection is maximized when the angle of the reflected light equals the angle of the incident light.

We use the vector h that lies "halfway" between e and l:

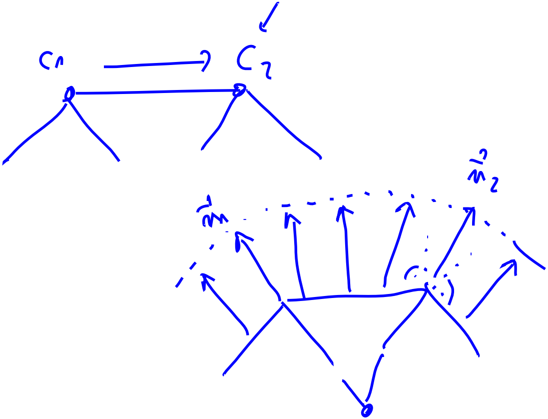
$$c = c_l c_p (h \cdot n)^p$$

$\sim \cos\theta$



Phong interpolation

Interpolating color values doesn't combine well with the Phong model: highlights in the interior of triangles are completely lost.

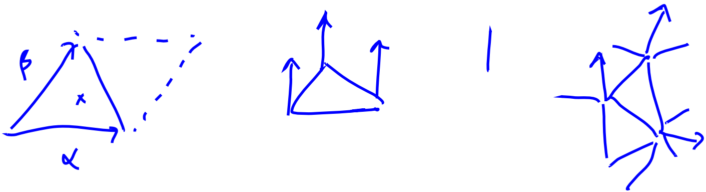


Phong interpolation

Interpolating color values doesn't combine well with the Phong model: highlights in the interior of triangles are completely lost.

However, we can interpolate normals instead, and do the computations of the Phong model using the interpolated normals.

$\vec{n} = \alpha \vec{n}_1 + \beta \vec{n}_2 + \gamma \vec{n}_3$, with $\alpha + \beta + \gamma = 1$



Global illumination

All the approaches so far, i.e.

- Diffuse shading
- Gouraud interpolation
- Phong shading
- Phong interpolation

calculate local illumination

Problem: all points with normals facing away from the light will be black

Not realistic! We need some sort of global illumination that also consideres, e.g. the reflection of light from other objects.

Global illumination

One approach (in ray tracing): trace rays recursively (you did that in the programming assignments)

Further approaches: two "tricks" to achieve global illumination:

- Place a dim light source at the eye's position
- Add an ambient term to the diffuse light calculation

Ambient shading

Basic idea: add a constant color term to the color of each object that simulates some sort of "global light", i.e.

$$c = c_r c_l \max(0, l \cdot n)$$ diffuse shading

now becomes

$$c = c_r (c_a + c_l \max(0, l \cdot n))$$

Overview

Today, we will finish "the basic stuff" (chapter 1-9 of the book):

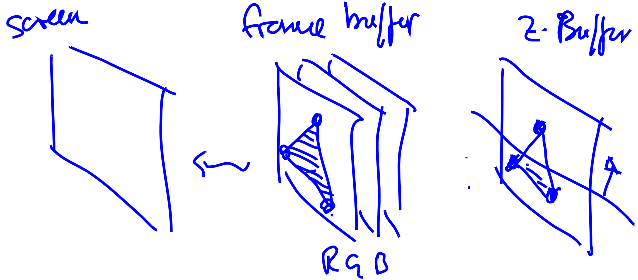
- We will *not* look into signal processing (chapter 4)
- We will look at triangle rasterization (cf. chapter 3, but differently covered here)
- Also at shading (cf. chapter 9, also some differences here)
- And again at Z-buffering (not covered in the book)

After the midterm exam, we will cover some "advanced topics":

- Ray tracing (chapter 10)
- Texture mapping (chapter 11)
- Graphics pipeline, part II (chapter 12)
- Radiosity and shadows (not in the book)

Z-buffering with scanline conversion

Recall that hidden surface elimination can be done efficiently with z-buffer algorithm. But how do we get the z-values for each pixel?



Again, we can compute these efficiently with bilinear interpolation, maintaining a Δ_z value for every edge, and computing Δ values for the z-increment along every scanline.

Thursday: MIDTERM EXAM

week: 40, datum: do 2-10-2008
tijd: 15.00-17.00 uur, zaal: JAARB-HAL 5

Content:
all material from the first five lectures
and the first three tutorials
(i.e. everything up to and including "transformations").

Note: no tutorials or labs on Thursday

Thursday: MIDTERM EXAM

week: 40, datum: do 2-10-2008
tijd: 15.00-17.00 uur, zaal: JAARB-HAL 5

Please ...

- come in time
- bring your student ID
- bring a pen

... and make it easy for us to correct
(i.e. write a lot of correct answers!)

Good luck and see you all Thursday!