Basic Techniques in Computer Graphics

Winter 2017 / 2018

The slide comments are not guaranteed to be complete, they are no alternative to the lectures itself. So go to the lectures and write down your own comments!

Recap

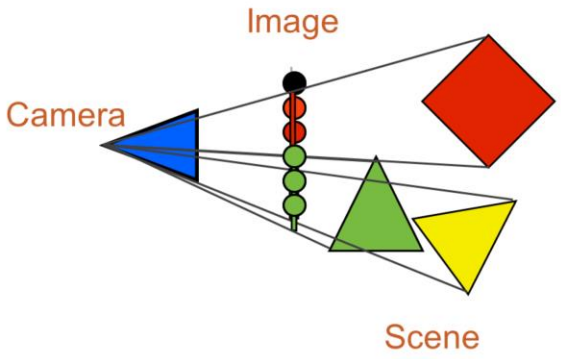
now we know

- how to represent geometry
  (points, lines, planes, polygons)
- how to transform it
  (linear maps, affine maps, extended coordinates)
- how to map it into camera coordinates
  (perspective maps, homogeneous coordinates)

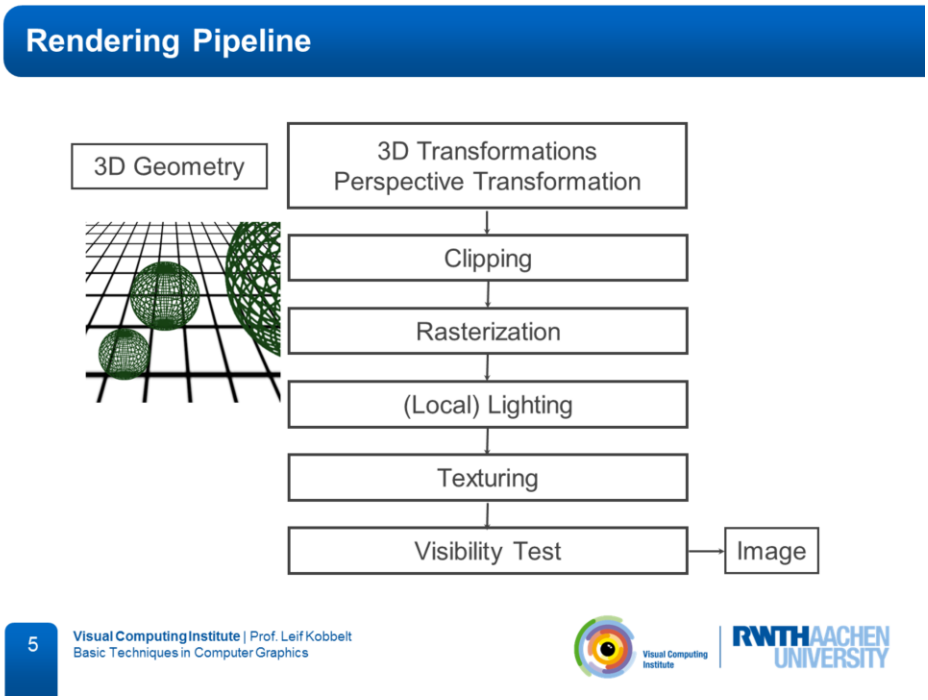
- next we'll learn
  - how to create a (perspectively correct) 2D image

This is what we learned in the last chapter. In this chapter we will learn how to create perspectively correct images. Color and lighting however will be discussed later.

Idea Rasterization

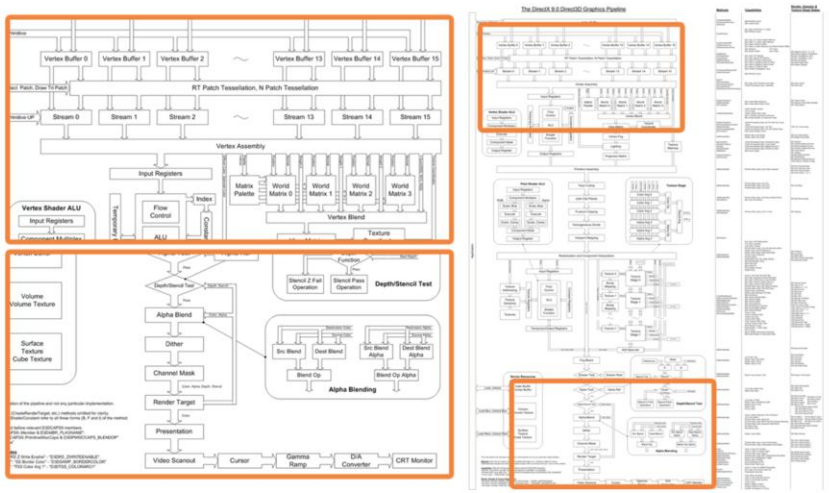


Rasterization works the other way around: each object gets projected onto the screen and the parts of the image that it covers will get tinted in its color.
We will learn later in this chapter how we can ensure that the green triangle does not get overpainted by the yellow one (visibility problem).
We will also learn how to take care of geometry that is only partially visible because its projected form does not lay inside of the screen (clipping).
Rasterization is a forward mapping technique as the objects are mapped forward to the camera.



One simplified example of a rasterization-based Rendering Pipeline. Some details change with newer graphics hardware generations or different implementations. For example the lighting can also be done before the rasterization. We will learn later what the advantages and disadvantages of each approach are.
3D Transformations: Move, Rotate, Scale the objects in a way, that they are in front of the camera as intended, the perspective transformations are done here also.
Clipping: remove everything that can‘t be seen because it‘s not in front of the camera.
Rasterization: calculate which pixels will „see“ which objects.
Lighting: calculate the local light.
Texturing: place images on the objects to simulate more details.
Visibility Test: make sure only the front most objects can be seen.

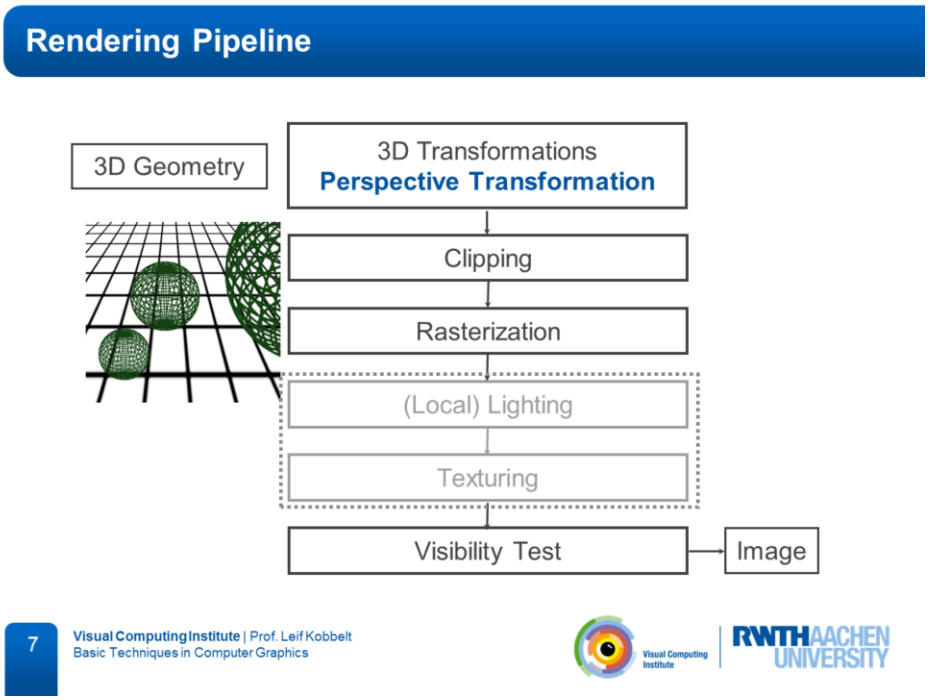
Direct3D 9 Rendering Pipeline



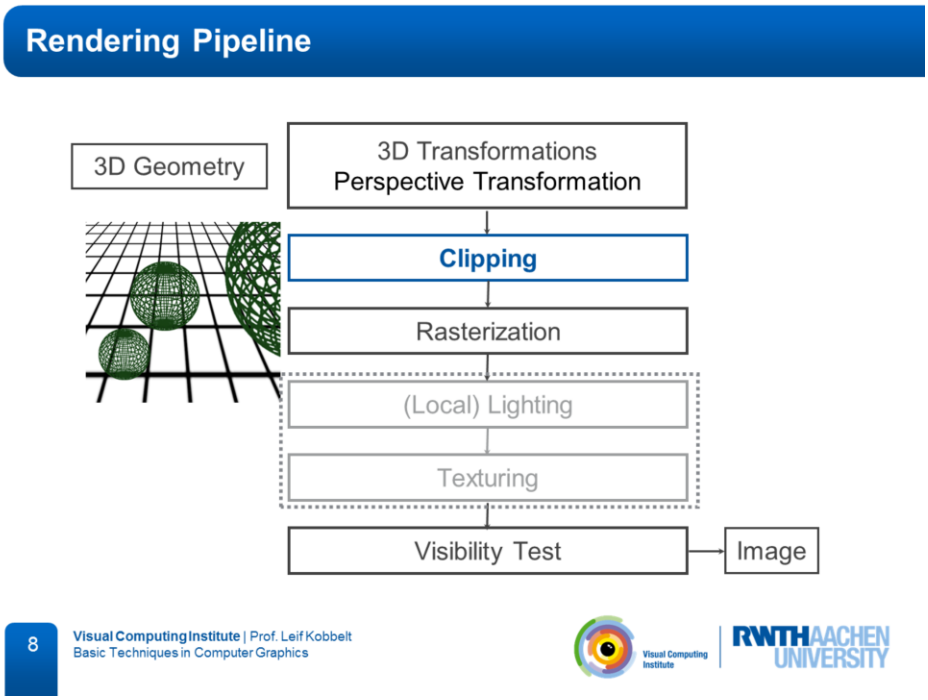
A real rasterization pipeline:
The Direct3D 9 pipeline looks quite overwhelmingly complex but this lecture will teach you all the basics to understand such diagrams. But don‘t worry, we won‘t go into that details!

This diagram was taken from
http://www.xmission.com/~legalize/book/preview/poster/index.html

What you should remember is this: The Rendering-Pipelines on our slides are just simplified models of whats going on in real GPUs. Some parts are also changing from time to time, that‘s why a Rendering-Pipeline in a book can look slightly different from our abstractions!



One simplified example of a rasterization-based Rendering Pipeline. Some details change with newer graphics hardware generations or different implementations. For example the lighting can also be done before the rasterization. We will learn later what the advantages and disadvantages of each approach are.
3D Transformations: Move, Rotate, Scale the objects in a way, that they are in front of the camera as intended, the perspective transformations are done here also.
Clipping: remove everything that can‘t be seen because it‘s not in front of the camera.
Rasterization: calculate which pixels will „see“ which objects.
Lighting: calculate the local light.
Texturing: place images on the objects to simulate more details.
Visibility Test: make sure only the front most objects can be seen.



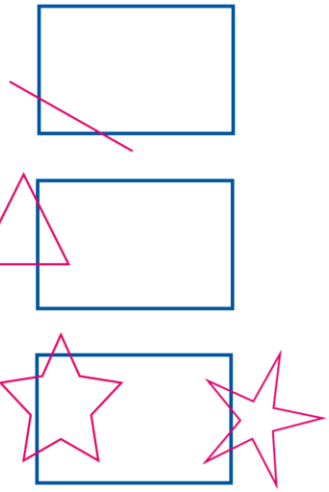
One simplified example of a rasterization-based Rendering Pipeline. Some details change with newer graphics hardware generations or different implementations. For example the lighting can also be done before the rasterization. We will learn later what the advantages and disadvantages of each approach are.
3D Transformations: Move, Rotate, Scale the objects in a way, that they are in front of the camera as intended, the perspective transformations are done here also.
Clipping: remove everything that can‘t be seen because it‘s not in front of the camera.
Rasterization: calculate which pixels will „see“ which objects.
Lighting: calculate the local light.
Texturing: place images on the objects to simulate more details.
Visibility Test: make sure only the front most objects can be seen.

Clipping

Line clipping

Triangle clipping

Polygon clipping

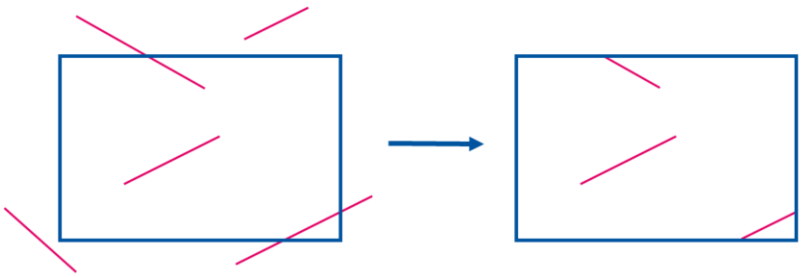


For each primitive type we know different algorithms for clipping. Also: here we are looking at the special case of clipping against a rectangle.

Remember: Polygon and triangle are often used to describe the same thing: Faces with exactly tree vertices. This is not correct, a triangle is a polygon but a polygon in general can have more than three vertices.

Line Clipping

- Clip against viewing rectangle
- Check in/out of endpoints



We wont go into details here as clipping is handled by the GPU for us and rasterization of non-triangles has become uncommon for real-time rendering.

Interested in line clipping? Look up the Cohen-Sutherland or Liang-Barsky algorithms.

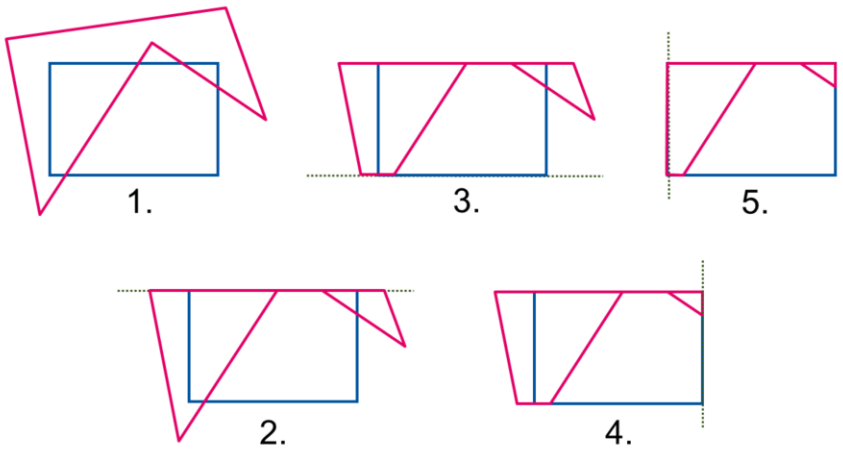
Outcode are used to determine if a line pixel is visible. Outcode consists of 4 bits, one for each clipping line. Bit is zero if pixel is outside regarding a clipping line, otherwise one. Pixel is inside viewing rectangle if all outcode bits are positive (1).

Outcodes for early rejection of lines, if start and end point of line are both outside with respect to one of the clipping lines.

Polygon Clipping

- Polygons may fall into several pieces
- March along polygon edges ...
  - Fully inside
  - Fully outside
  - Leaving
  - Entering
- Sutherland-Hodgmann
  clip with respect to a sequence of half-spaces
- Liang-Barsky
  clip once, add turning points if necessary

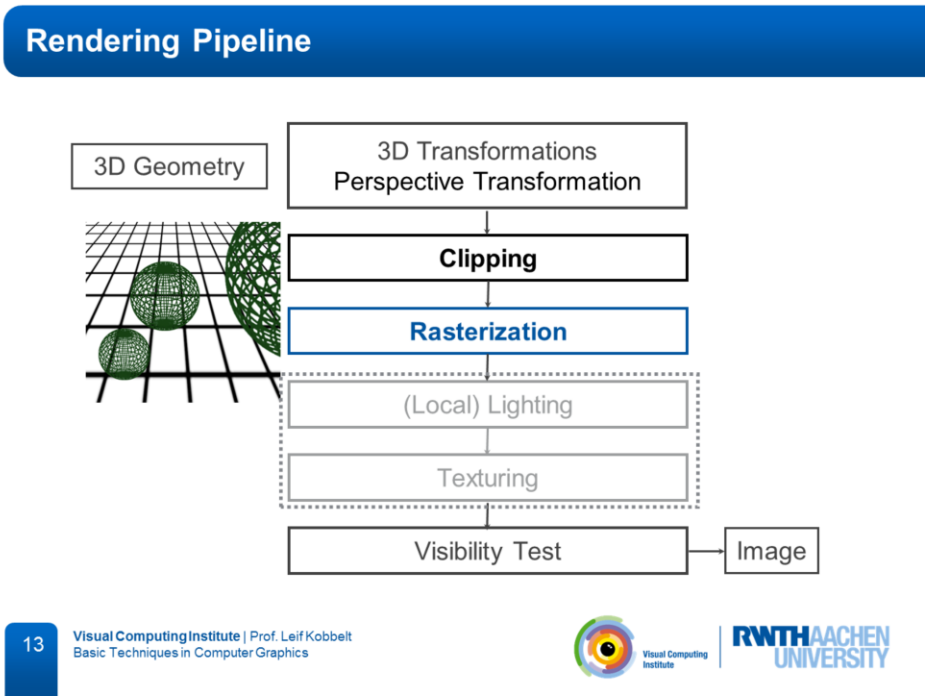
Sutherland-Hodgman



This algorithms can clip arbitrary polygons: This is done in multiple steps.

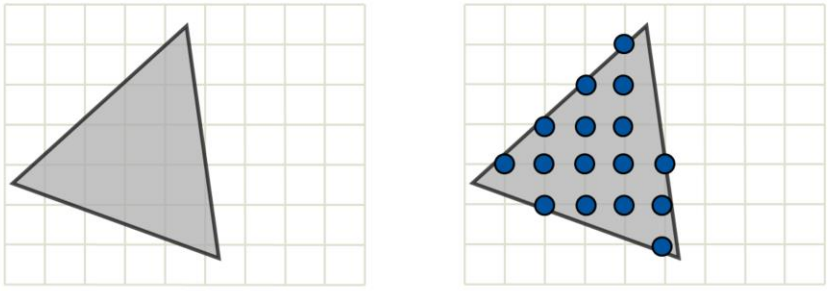
We have much less special cases if we would only handle triangles instead of arbitrary polygons. That‘s one reason why graphics hardware internally only works on triangles!

Introduce turning points to handle clipping around corners of the clipping rectangle.



One simplified example of a rasterization-based Rendering Pipeline. Some details change with newer graphics hardware generations or different implementations. For example the lighting can also be done before the rasterization. We will learn later what the advantages and disadvantages of each approach are.
3D Transformations: Move, Rotate, Scale the objects in a way, that they are in front of the camera as intended, the perspective transformations are done here also.
Clipping: remove everything that can‘t be seen because it‘s not in front of the camera.
Rasterization: calculate which pixels will „see“ which objects.
Lighting: calculate the local light.
Texturing: place images on the objects to simulate more details.
Visibility Test: make sure only the front most objects can be seen.

Rasterization



continuous primitive discrete fragments

Rasterization is the process of creating fragments at positions where a primitive intersects the pixel grid.

Rasterization

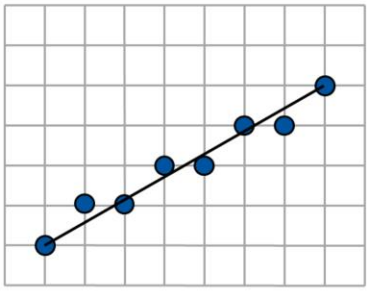
Fragment != Pixel

- A fragment is a „pixel candidate“
- A fragment can get discarded and will never become a pixel
- A pixel can be the mixture of multiple fragments

The rasterization stage creates fragments, those can get discarded for example if the fragment will not be visible in the final image (its hidden by other polygons). On the other hand a fragment can be mixed into an existing pixel instead of replacing it (think of color blending used for transparent objects).

Line Rasterization

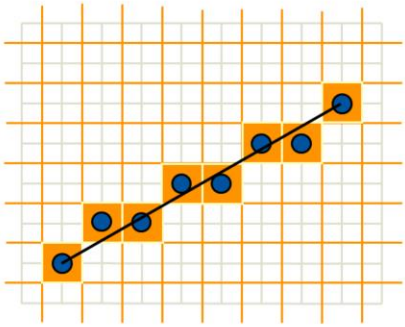
Continuous line ↔ discrete pixel grid



We will now look at rasterization of lines as one simple example of rasterization. There are special algorithms for rasterizing more complex primitives but this lecture won‘t go into those details.

Line Rasterization

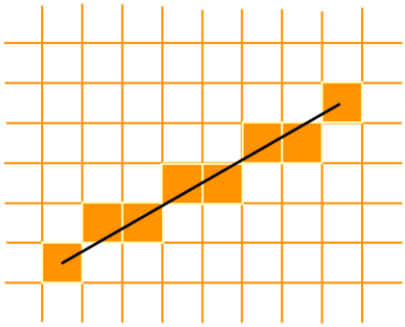
Continuous line ↔ discrete pixel grid



If we would draw every pixel touched by the line, it would not result in a thin, „visually appealing“ line

Line Rasterization

Continuous line ↔ discrete pixel grid



For every discrete pixel coordinate in X direction we can calculate the Y coordinate and color that pixel iff the line lays in the first quadrant of the coordinate system. If it lays somewhere else we have to switch signs and ‚mirror' the algorithm.

Line Rasterization

Endpoints $[x_0, y_0]^T$, $[x_1, y_1]^T$

Assumption: slope $\in [0,1]$ and $x_0 < x_1$

Parametric:

- $y = mx + t$, $m=(\Delta y / \Delta x) = (y_1-y_0) / (x_1-x_0)$

Implicit:

- $F(x,y) = ax + by + c$
- $a = \Delta y$, $b = -\Delta x$, $c = \Delta x\, t$
- $\{ [x, y]^T \in R \mid F(x,y) = 0 \}$

First Guess

BAD: multiplication + rounding

```
for (x=x0; x <= x1; ++x)
    set_pixel(x, round(m*x + t));
```


While this works, its more complicated than needed, because we need float operations here. Line rasterization can also be done with only integer operations.

DDA

Digital Differential Analysis
Incremental algorithm (curr. pixel $[x_i, y_i]^T$)
$y_{i+1} = y_i + m\,\Delta x = y_i + m \quad (\Delta x = 1)$

```
for (x=x0, y=y0; x <= x1; ++x, y+=m)
    set_pixel(x, round(y));
```

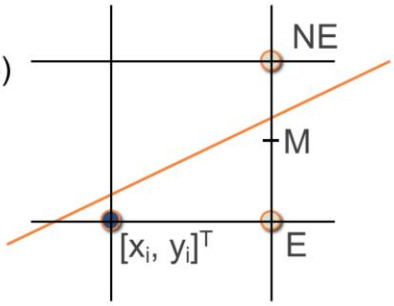
GOOD: no multiplication
BAD: rounding, y and m are floats

Only a single addition to compute y_i+1 from y_i, more efficient than multiplication/solving linear systems

Bresenham / Midpoint

Incremental algorithm (curr. pixel $[x_i, y_i]^T$)
Next pixel: **E** or **NE** (line below/above **M**)

- *Decision variable*:
  - **d** = F(M)
    = $F(x_i+1, y_i+0.5)$
  - **d** > 0 ? **NE** : **E**

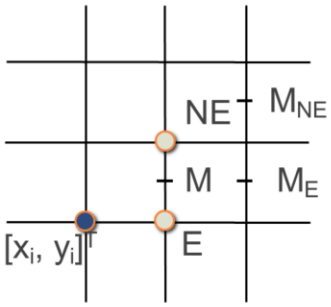


E: east, just go to the right
NE: north east: go to the right and one up

Other candidates are not possible, since we restrict the slope to [0,1].

Incremental update of d

- **E** chosen:
  - $d_{new} = F(x_i+2, y_i+0.5)$
    $= d_{old} + a$
  - $\Delta_E = a = \Delta y$
- **NE** chosen:
  - $d_{new} = F(x_i+2, y_i+1.5)$
    $= d_{old} + a + b$
  - $\Delta_{NE} = a + b = \Delta y - \Delta x$



Bresenham / Midpoint

- Initialization of d:
  - $d = F(x_0+1, y_0+0.5) = \Delta y - \Delta x / 2$
- Bad: $\Delta x / 2$ may not be integer
- Use $2 \cdot F(x,y)$ instead of $F(x,y)$
  $\Rightarrow$ **d**, Δ_E, Δ_{NE} are doubled
  $\Rightarrow$ sign of **F** and **d** unchanged

To get to a pure integer algorithm we have to get rid of the 1/2 term: so just scale all variables by 2.

Bresenham / Midpoint

```
Δx = x1 - x0; Δy = y1 - y0;
d = 2 Δy - Δx;
Δ_E = 2 Δy;
Δ_NE = 2 (Δy - Δx);

set_pixel(x_0, y_0);

for (x = x_0, y = y_0; x < x_1;)
{
    if (d < 0)  { d += Δ_E;   ++x;      }
    else        { d += Δ_NE;  ++x; ++y }
    set_pixel(x, y);
}
```

GOOD:
only integer
arithmetic !

The Bresenham algorithm is just an example of DDA. The same ideas can be extended to the rasterization of arbitrary polygons.

General Concept

- differential analysis can be used to evaluate polynomials of arbitrary degree
  - $F(x)$ has degree n
  - $\Delta F(x) = F(x+h) - F(x)$ has degree $n-1$
- lines: linear decision function
  constant update
- circles: quadratic decision function
  linear update

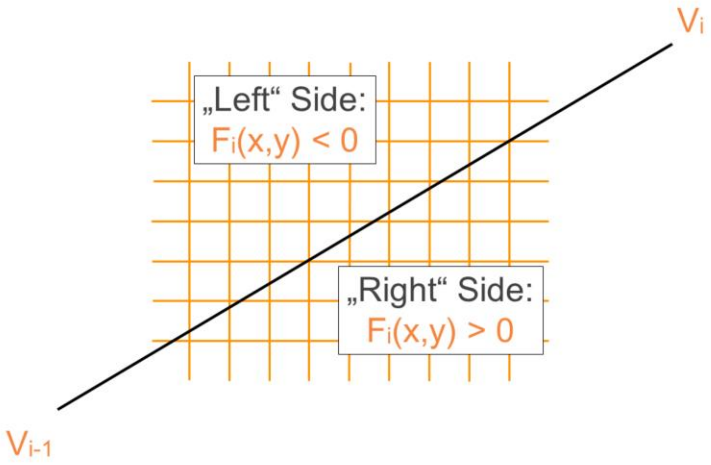
Evaluate linear updates recursively by constant updates.
An update of a polynomial of degree n can be turned into n additions by recursive DDA.

Polygon Rasterization: Pineda

- *A Parallel Algorithm for Polygon Rasterization* (Pineda'88)
- Convex polygon rasterization
  - Vertices $V_i = (X_i, Y_i)$, $0 \le i \le n$, $V_0 = V_n$
  - Edges E_i, $1 \le i \le n$ with end points V_{i-1} and V_i
  - Implicit function $F_i(x,y)$ for each edge E_i
- Sign of $F_i(x,y)$ related to position of (x,y) relative to edge E_i:
  - $F_i(x,y) < 0 \Rightarrow (x,y)$ to the „left" side of E_i
  - $F_i(x,y) = 0 \Rightarrow (x,y)$ is exactly on E_i
  - $F_i(x,y) > 0 \Rightarrow (x,y)$ to the „right" side of E_i

Paper: "A Parallel Algorithm for Polygon Rasterization", Juan Pineda, 1988

Polygon Rasterization: Pineda



Rasterize single lines with pineda by replacing the line by a thin rectangle. (Done in modern graphics hardware, no Bresenham => single rasterization unit for rasterization of triangles AND lines).

Polygon Rasterization: Pineda

- W.l.o.g. let $V_0,\ldots,V_n$ be defined in counter-clockwise order

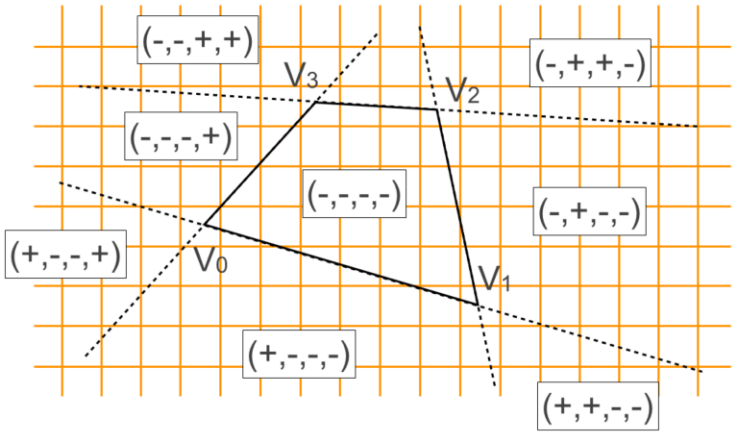
$\Rightarrow$A fragment (x,y) is interior to a polygon if it is to the „left“ side of each polygon edge E_i:

$$F_i(x,y) \leq 0, \text{ for all } 1 \leq i \leq n$$

- Compute tupel $(F_1(x,y),\ldots, F_n(x,y))$ for each fragment in candidate area around polygon (e.g. in bounding box)

Basic Idea: Ensure consistent vertex order, then the interior of the polygon should be on the same side (left for counterclockwise, right for clockwise order) of the polygon.
Not only compute/interpolate color and depth for each fragment, but also for each F_i

Polygon Rasterization: Pineda



Four edges (counter-clockwise):
E_1 (end points V_0 & V_1)
E_2 (end points V_1 & V_2)
E_3 (end points V_2 & V_3)
E_4 (end points V_3 & V_4=V_0)

=> each fragment stores tupel (F_1, F_2,F_3,F_4)

fragment is in interior of polygon with respect to edge E_i if F_i is negative

Pineda Rasterization works for arbitrary non-concave polygons.

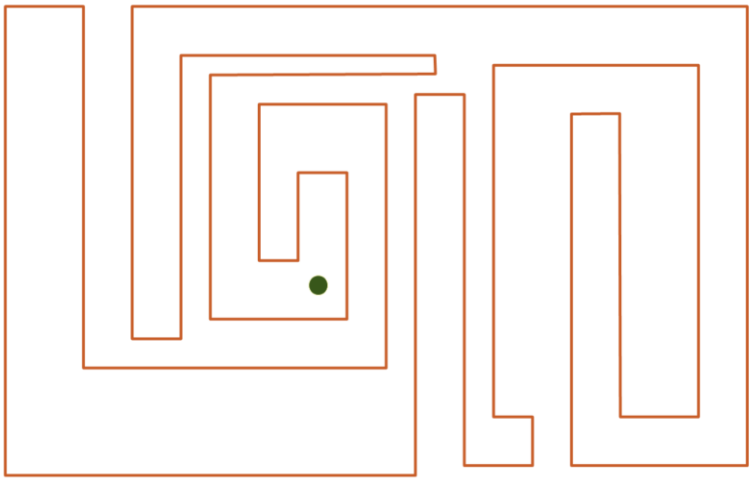
Optimizations

Parallelization
- Recall: $F_i(x,y) = ax + by + c$
- ➞ $F_i(x+s,y+t) = F_i(x,y) + as + bt$
- Compute $F_i(x,y)$ for block of fragments simultaneously!

Clipping
- Don‘t traverse fragment that lie beyond the projections of vertical & horizontal clipping lines
- ➞ Explicit clipping required only for near/far plane

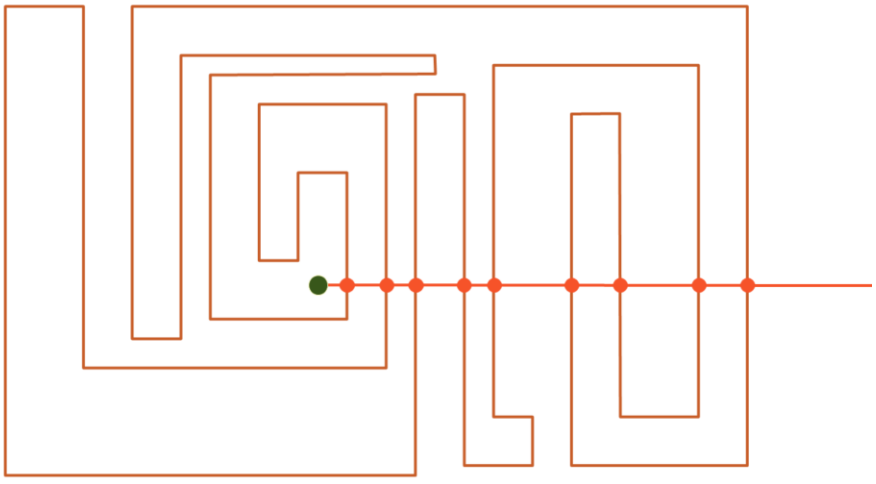
Since $F_i(x+s,y+t) = F_i(x,y) + as + bt$, one can compute $F_i(x+s,y+t)$ simply from $F_i(x,y)$ (interpolation)

Concave Polygons

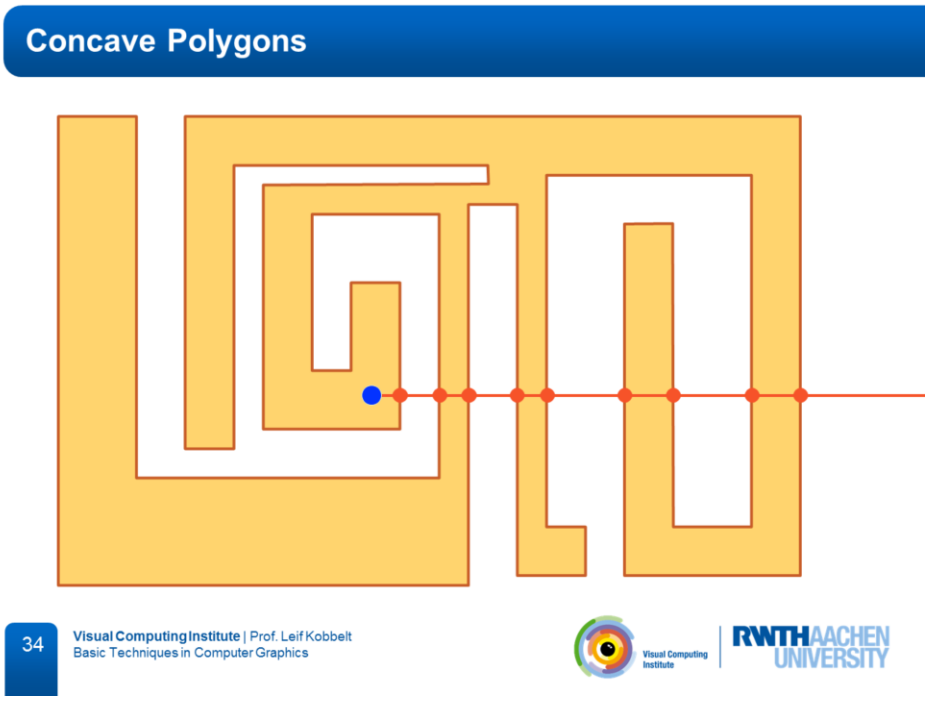


The blue dot is one fragment and it has to be determined whether it is covered by this polygon.

Concave Polygons



One way to check this is to shoot a ray in one direction to infinity and count the intersections. If the number of intersections is odd, the point is part of the polygon, otherwise it lays outside.
Again, this test would be much easier if we only would have to rasterize triangles.



Scanline Conversion: Scanline intersections give spans of pixels that lie in the inside of the polygon

Polygon Triangulation

- *Marching* (corner cutting)
  - determine orientation
    - 2D cross product: $u \times v = u_x v_y - u_y v_x$
    - p_i on conv. hull: $(p_{i+1} - p_i) \times (p_{i-1} - p_i)$
    - e.g. p_i with smallest x-coordinate
  - cycle through vertices
  - chop off *convex* corners
  - test against all *concave* corners

Working only with triangles is much easier than taking care of arbitrary polygons: all vertices are laying on one plane, clipping and rasterizing gets simpler etc.
In order to work with arbitrary polygons the application might have to triangulate the polygons if it still wants to let the user work with arbitrary polygons.

The basic idea is to cut away corners. But we have to check if the corner is convex or concave, because we can only cut away convex corners. The check can be done by calculating the angle between the vectors and check the sign.

Angle large than 180°: concave corner, otherwise convex corner

For 2D coordinates, add third coordinate that is zero, then the cross product is (0,0,u_x*v_y-u_y*v_x)^T.

To determine the sign of convex corners, compute cross product for corner from which we know that it is convex, for example the leftmost corner.

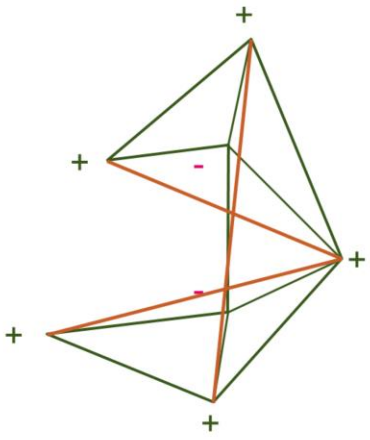
Special case: 3 Points on a line! The corner is consider to be concave.

Polygon Triangulation

- for each corner vertex P_i
  - check if (P_{i-1}, P_i, P_{i+1}) is convex
  - check if any concave corner P_j lies within the triangle (P_{i-1}, P_i, P_{i+1})
  - remove P_i from the sequence of vertices

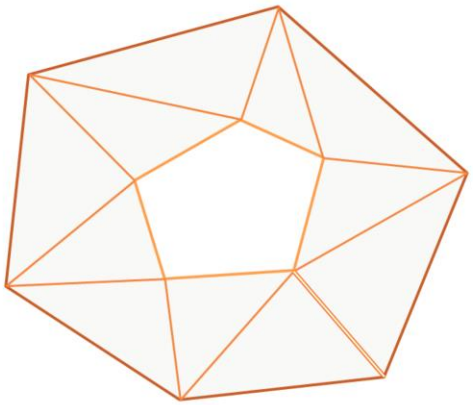
First, a check has to be performed whether a corner is convex. If so, we still can‘t cut it off right away because it does not necessarily completely lay within the polygon. Only if it does we can cut it off.

Polygon Triangulation



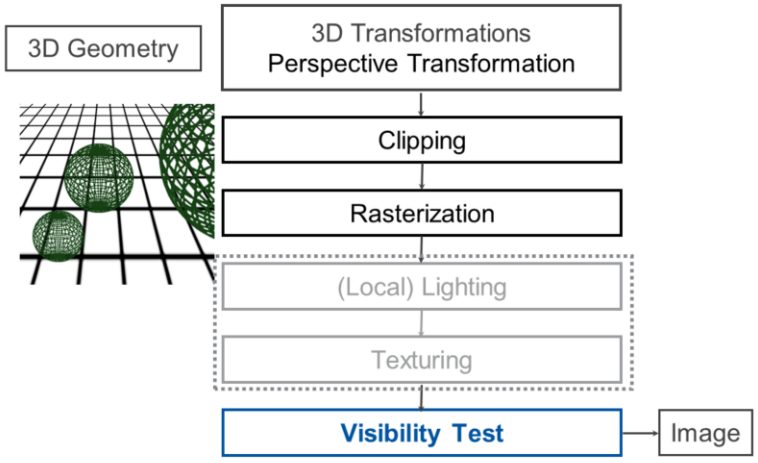
The signs here are the signs of the cross product. To tell whether positive or negative values correspond to convex corners, check the sign of the left-most edge (because that is guaranteed to be convex.
If a cut would be outside of the polygon as shown here, another corner has to be found.

Polygon Triangulation

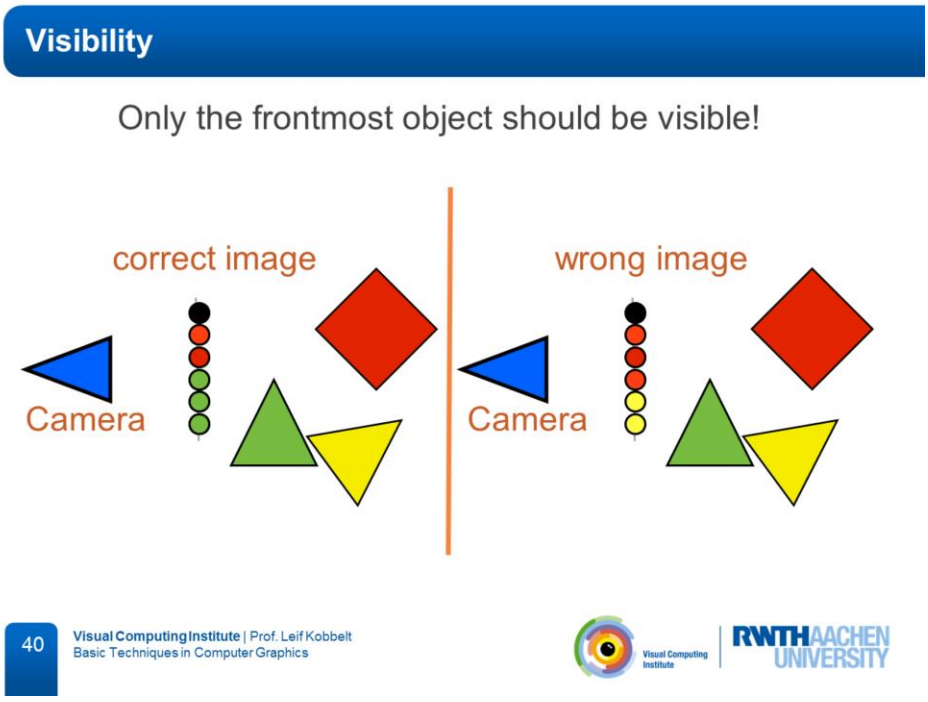


To triangulate a polygon with a hole in it, it has to be cut open first.

Rendering Pipeline



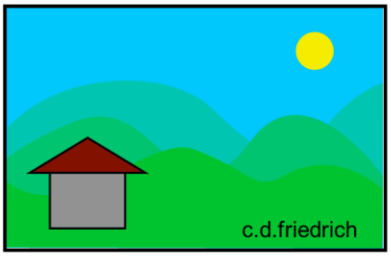
One simplified example of a rasterization-based Rendering Pipeline. Some details change with newer graphics hardware generations or different implementations. For example the lighting can also be done before the rasterization. We will learn later what the advantages and disadvantages of each approach are.
3D Transformations: Move, Rotate, Scale the objects in a way, that they are in front of the camera as intended, the perspective transformations are done here also.
Clipping: remove everything that can‘t be seen because it‘s not in front of the camera.
Rasterization: calculate which pixels will „see“ which objects.
Lighting: calculate the local light.
Texturing: place images on the objects to simulate more details.
Visibility Test: make sure only the front most objects can be seen.



green triangle is in front of red and yellow polygons => should be rendered last.

Painter´s Algorithm

Painter‘s way of drawing a scene:



One naive way to ensure a correct rendering is the Painter‘s Algorithm:
Like a painter we start rendering the objects (to be precise: the polygons) back to front. This way polygons in front will overdraw the polygons more far away.

Painter´s Algorithm

Sort all polygons w.r.t. z_{max} (farthest)

Paint polygons from back to front

Disambiguate visibility if *z*-ranges of polygons are *overlapping*

One naive way to ensure a correct rendering is the Painter‘s Algorithm:
Like a painter we start rendering the objects (to be precise: the polygons) back to front. This way polygons in front will overdraw the polygons more far away.

Painter´s Algorithm

Disambiguate visibility: $z_{max}(A) > z_{max}(B) > z_{min}(A)$

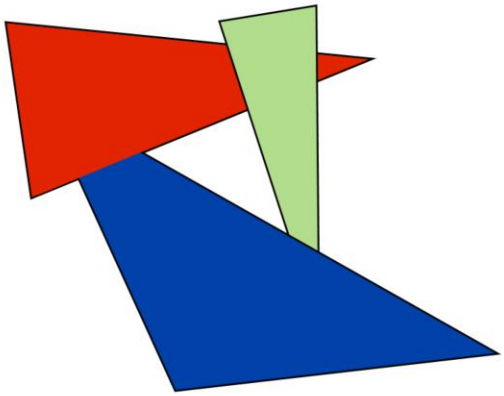
- [x,y]-ranges not overlapping ?
- A behind supporting plane of B ?
- B in front supporting plane of A ?
- Non-overlapping projections?

paint A

- B behind supporting plane of A ?
- A in front supporting plane of B ?

swap A & B

Painter´s Algorithm



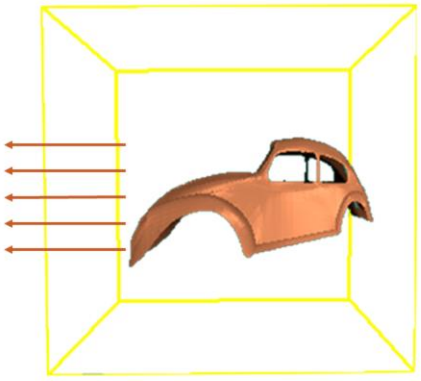
There are situation where no correct draw order can be found, in that case one triangles has to get split up.

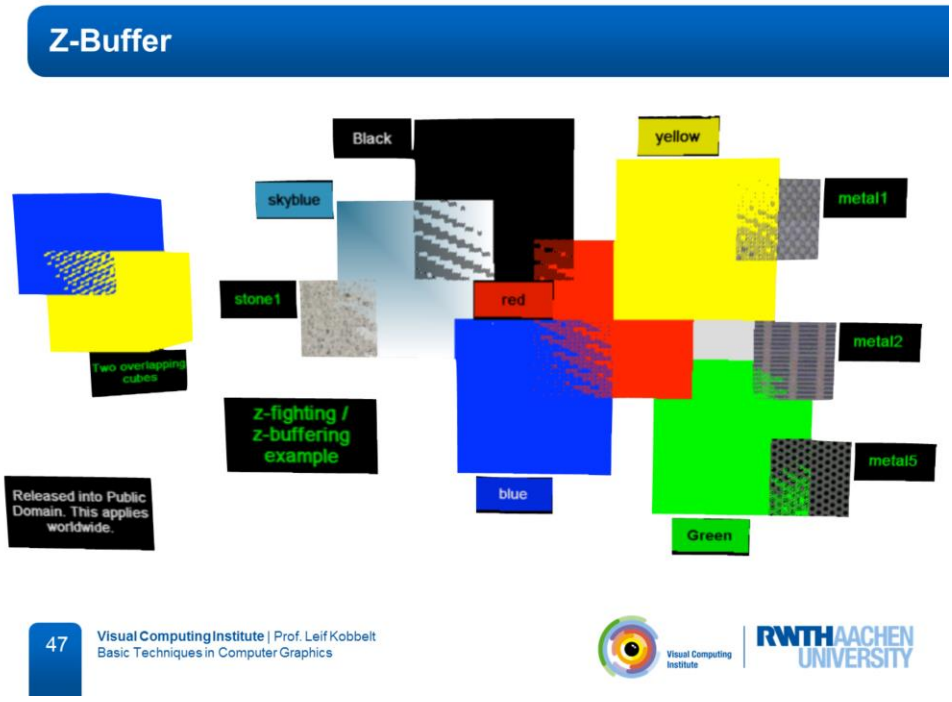
Z-Buffer

- Remember for each Pixel the distance to the eye of the painted object
- If a new Pixel is closer
  - Overdraw
  - Discard the Pixel otherwise

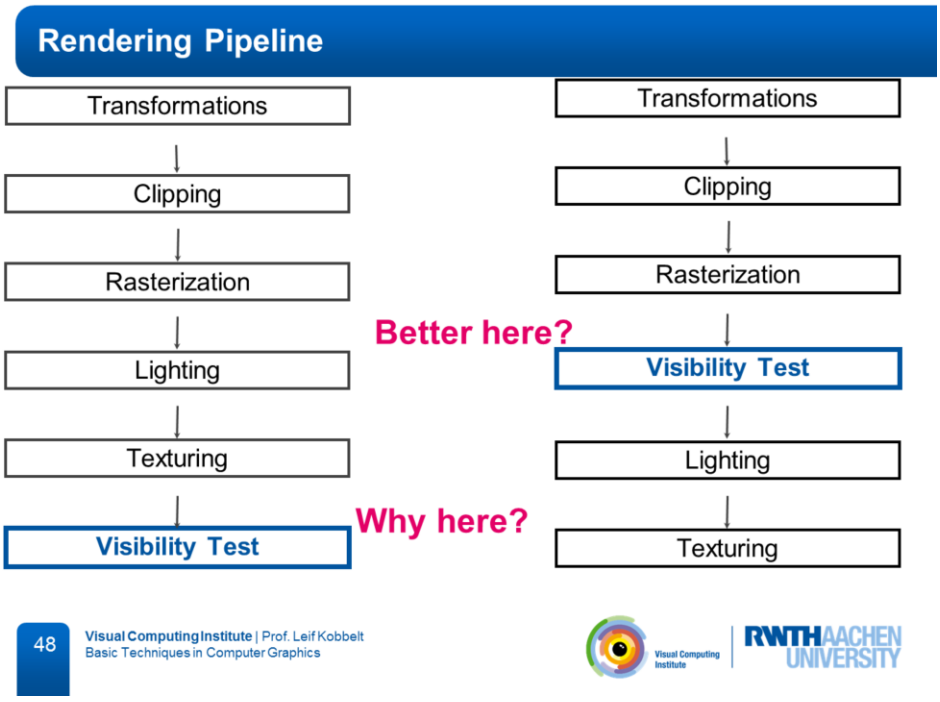
The basic idea is roughly the same as with ray-tracing. We have to search for the closest object per pixel of the final image. But with rasterization the image gets drawn primitive (normally == triangle) by primitive and not pixel by pixel. Thats why we have to save the distance of the previously „found“ or drawn pixels for all possible pixels of the screen. This array is known as the Z-Buffer and often stores the distance as a 24bit float value in modern GPUs.

Frustum Transform

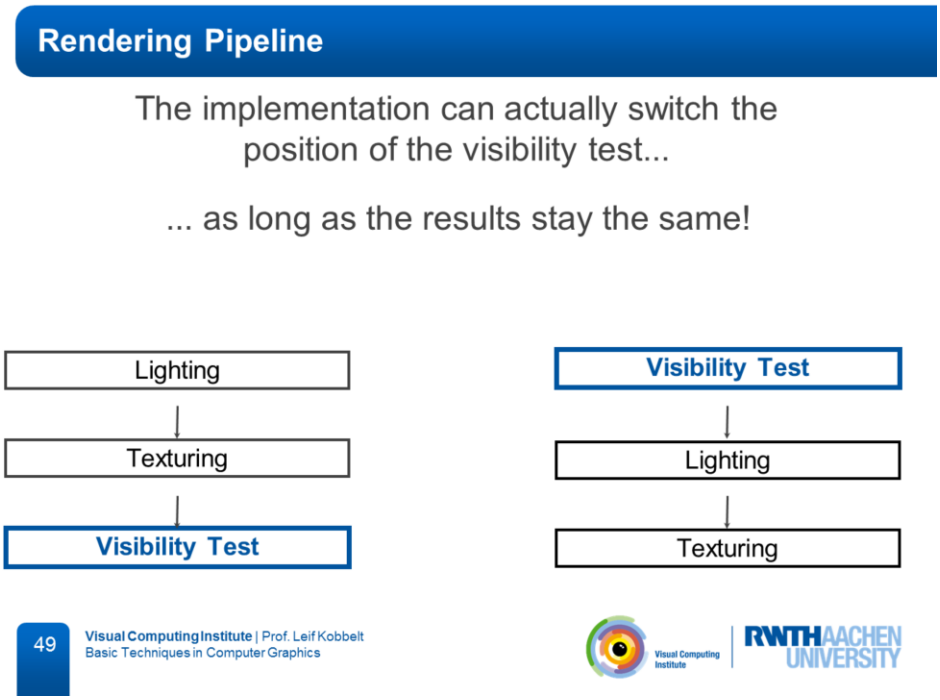




Examples for z-fighting



Why is the visibility test performed after the lighting and texturing? Wouldn‘t an early test save a lot of computations?



The OpenGL implementation has to behave as if the visibility test was done at the end of the pipeline but can in fact do it earlier. However, in some situations it is not guaranteed that the results would be the same and then the early Z-test is not allowed.
Lighting and texturing are done in fragment shaders nowadays and those are allowed to modify the depth value. In OpenGL 4.2 shaders can also write to textures and atomic counters. So even if a fragment is not visible because of the depth test, it can create a side effect. In such cases an early Z-test can not be performed.

What‘s missing

- Light simulation
  - highlights, shadows...
- Color
  - textures, material

Color, lighting, texturing will get covered in chapters 5&6. But first we want to look into the practical parts and learn something about rendering APIs.