Graphics 2008/2009, period 1

Lecture 7

Hidden surface removal

In case you didn't notice ...

... half of the lectures are done! Time to sit back and relax, look into the future, and reflect on the past.

Ok, let's sit back and relax first ...

Lifted (copyright: Pixar/Disney), source: YouTube, http://www.youtube.com/watch?v=1NO2MbkIpxo

In case you didn't notice ...

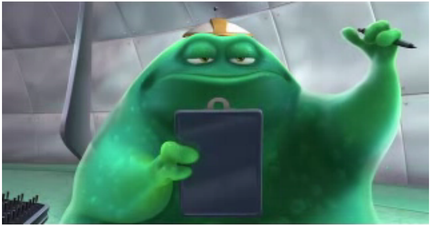
... half of the lectures are done! Time to sit back and relax, look into the future, and reflect on the past.

Now let's look into the future. What's next?

In case you didn't notice ...

... half of the lectures are done! Time to sit back and relax, look into the future, and reflect on the past.

Now let's look into the future. What's next?



The midterm exam!

Thu, Oct 2, 2008
15.00-17.00 h
Zaal: JAARB-HAL 5

For time and location info check
http://www.cs.uu.nl/education/vak.php?vak=INFOGR

In case you didn't notice ...

... half of the lectures are done! Time to sit back and relax, look into the future, and reflect on the past.

Now let's look into the future. What's next?

Some organizational remarks:

- come in time
- bring a pen (no pencil)
- and your student id

Note: You may not use books, notes, or any electronic equipment (including cell phones!).

In case you didn't notice ...

... half of the lectures are done! Time to sit back and relax, look into the future, and reflect on the past.

Now let's look into the future. What's next?

If you did what I said, i.e.

- attended / watched the lectures,
- did the exercises, and
- prepare appropriately

you should do fine.

In case you didn't notice ...

... half of the lectures are done! Time to sit back and relax, look into the future, and reflect on the past.

The past: what did we learn so far?

- Introduction and ray tracing (lecture 1)
- Vectors and curves (lecture 2)
- Curves, surfaces, and shading (lecture 3)
- Matrices and determinants (lecture 4)
- Linear and affine transformations (lecture 5)
- Perspective projection and graphics pipeline, part I (lecture 6)

Exam: lecture 1-5 and tutorials 1-3

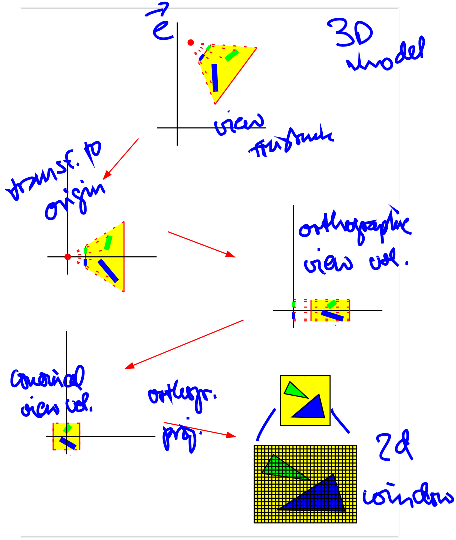
Current state of affairs

So far, we have only seen the projection phase of projection-based rendering.

What's missing?

matrix multiplication(s)

$$M = M_o M_p M_v$$

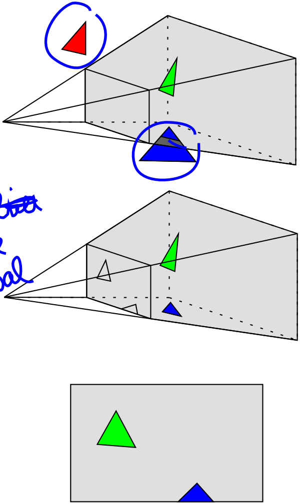


Stages in the graphics pipeline

We distinguish several stages in the graphics pipeline

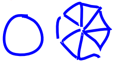
- Triangles that lie (partly) outside the view frustum need not be projected, and are clipped.
- The remaining triangles are projected if they are front facing.
- Projected triangles have to be shaded and/or textured.

hidden surface removal

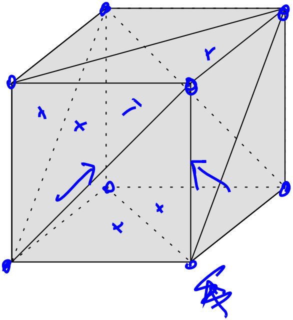


Backface culling

Surfaces of polyhedral objects are often modeled with connected, outward facing triangles.



Q: How many triangles do we need to model a cube? What is the maximum number of these triangles that is visible from any given viewing direction?

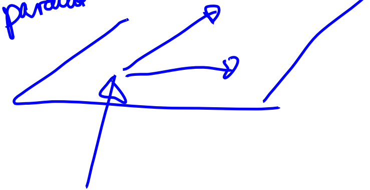


Backface culling

Obviously, if a camera faces the backside of a triangle, there is no need to draw it.

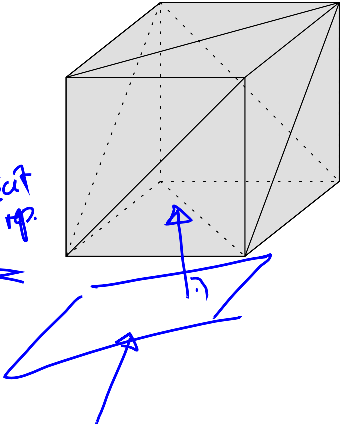
But how do we know which side we are looking at?

parametric rep.



implicit rep.

=



Implicit equations revisited

Given a normal vector of a plane, what was the implicit equation of the plane again?

$F(x,y,z) = ax + by + cz + d = 0 \quad (1)$

$\vec{p}_0, \vec{p}_1, \vec{p}_2$ on plane F

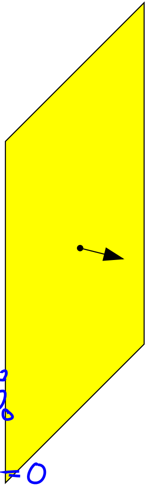
$\vec{n} := (\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)$

$\vec{n}$ in (1): $d = -\vec{n} \cdot \vec{p}_0$

Vect. repr.:

$f(\vec{p}) = \vec{n} \cdot \vec{p} - \vec{n} \cdot \vec{p}_0$

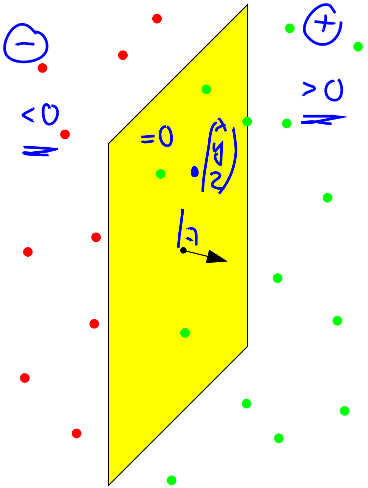
$= \vec{n}(\vec{p} - \vec{p}_0) = 0$



Implicit equations revisited

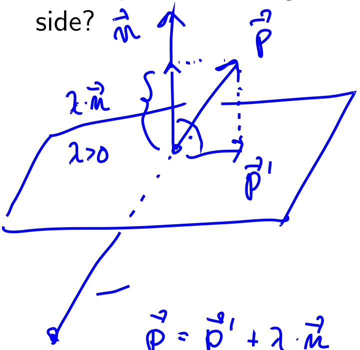
What do we get if we evaluate the plane equation for points on the side of the plane to which the normal points?

$f(x,y,z)=0$



Implicit equations revisited

Does the normal vector point to the positive or negative side?



$$\vec{p} = \vec{p}\,' + \lambda \cdot \vec{n}$$

$$f(x,y,z) = ax + by + cz + d$$

$$f(\vec{p}) = \vec{n} \cdot \vec{p} + d, \quad \vec{n} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

$$f(\vec{p}) = \vec{n}(\vec{p}\,' + \lambda \vec{n}) + d$$

$$= \underline{\vec{n}\,\vec{p}\,'} + \lambda \vec{n}^2 \underline{+ d}$$

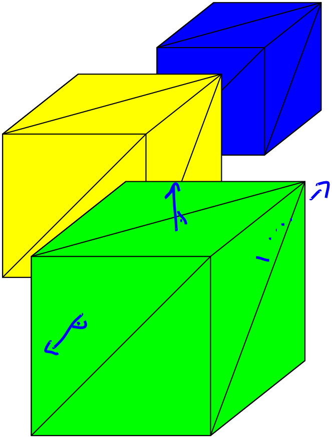
$$= \underbrace{f(\vec{p}\,')}_{=0} + \underbrace{\lambda}_{>0} (\underbrace{a^2}_{\geq 0} + \underbrace{b^2}_{\geq 0} + \underbrace{c^2}_{\geq 0}) > 0$$

$\geq$

Backface culling

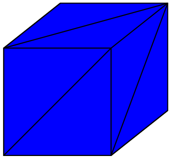
For backface culling we just need to know if the camera is on the positive or negative side of the plane / triangle.

But how do we eliminate, e.g. hidden lines from overlapping objects?



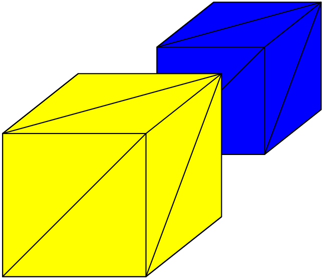
Painter's algorithm

If we draw triangles that are further away before triangles that are closer to the viewing point, we end up with a correct image:



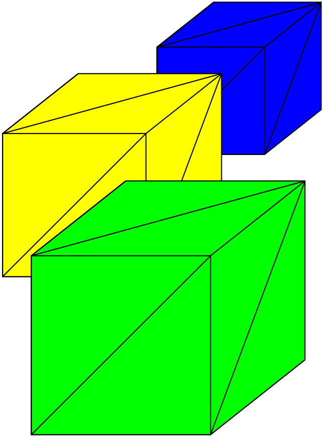
Painter's algorithm

If we draw triangles that are further away before triangles that are closer to the viewing point, we end up with a correct image:



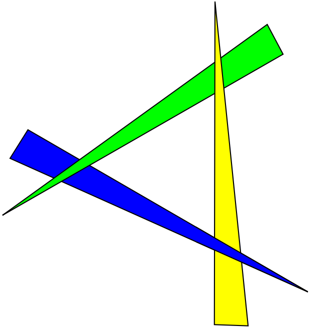
Painter's algorithm

If we draw triangles that are further away before triangles that are closer to the viewing point, we end up with a correct image:



Cyclic overlap

However, not every arrangement of triangles admits a back-to-front ordering:



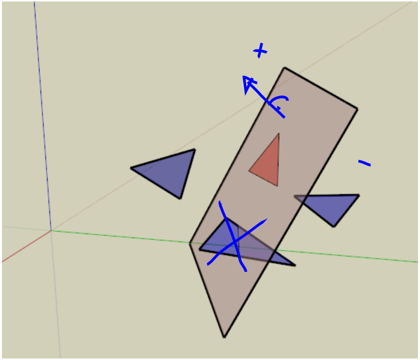
Most common approaches

- Binary space partitioning trees
  uses a preprocessed data structure encoding the position and order of objects

- Z-buffer
  uses additional storage for depth information (mostly hardware, but implementations in software exist as well)

BSP-trees: basic idea

Initial simplification: assume two triangles T_1 and T_2 and no triangle crosses the plane defined by the other one.

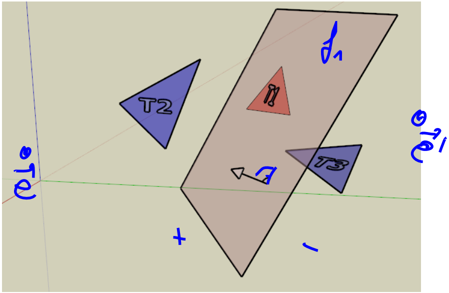


BSP-trees: basic idea

Assume f_1 is the plane defined by triangle T_1 and e is the eye position.

IF $f_1(e) > 0$ THEN
  draw T_3, T_1, T_2

IF $f_2(e) < 0$ THEN
  draw T_2, T_1, T_3s



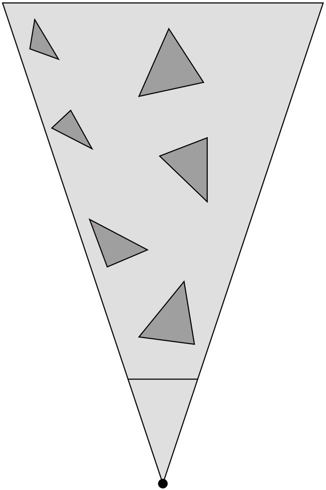
We can also generalize to more than 3 triangles

BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur... and draw the triangles in the proper order.

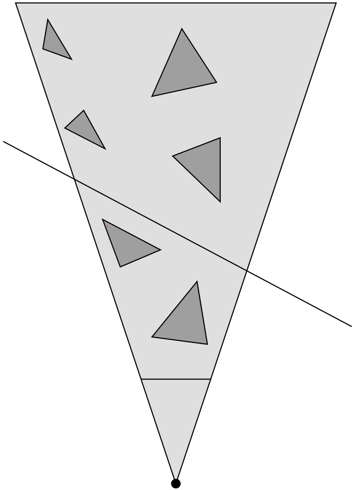


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur. . . and draw the triangles in the proper order.

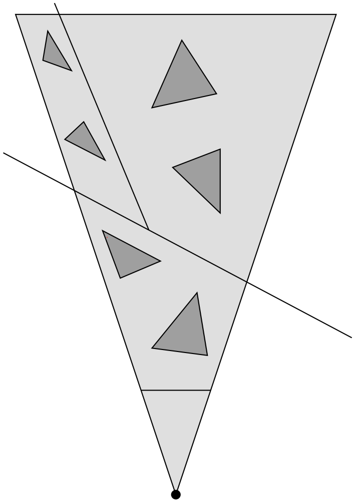


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur… and draw the triangles in the proper order.

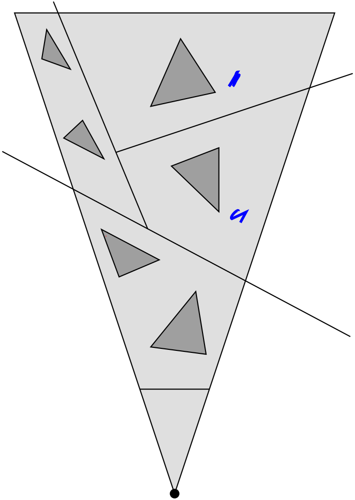


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur. . . and draw the triangles in the proper order.

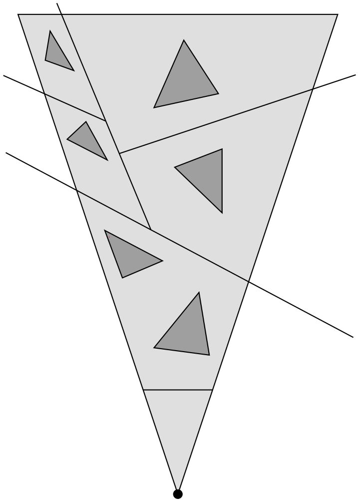


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur... and draw the triangles in the proper order.

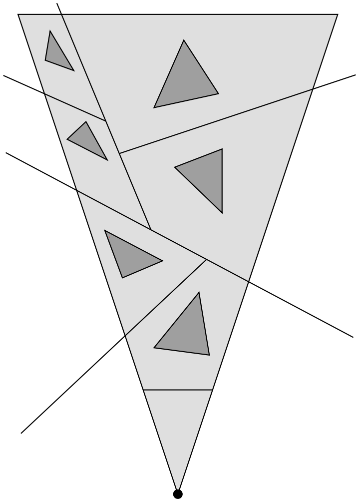


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur... and draw the triangles in the proper order.

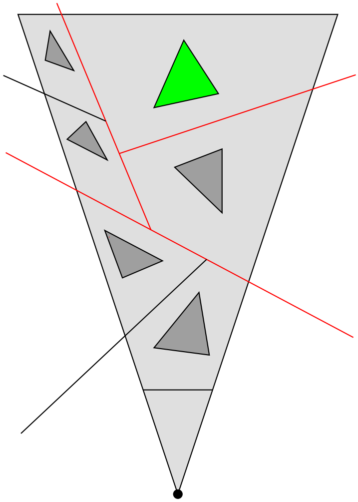


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur... and draw the triangles in the proper order.

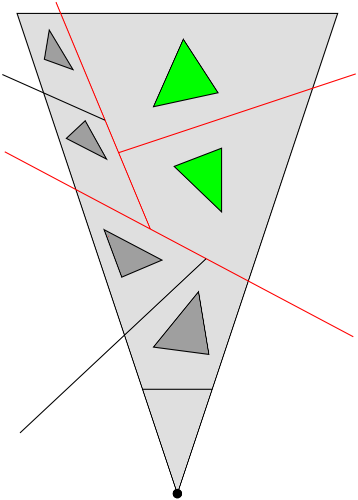


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur. . . and draw the triangles in the proper order.

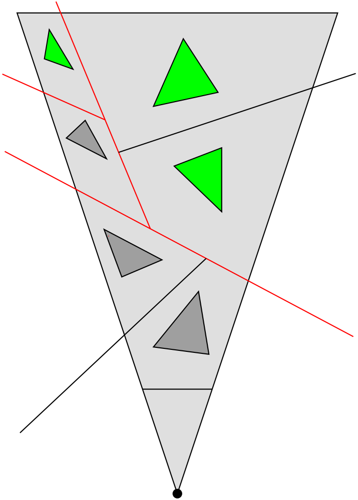


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur. . . and draw the triangles in the proper order.

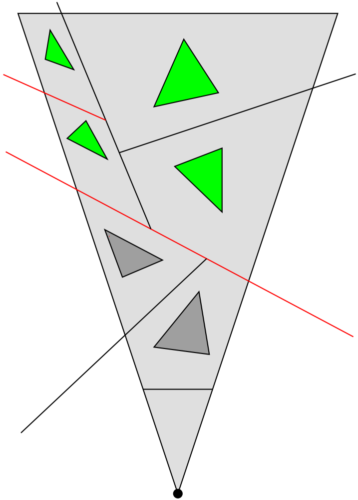


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur. . . and draw the triangles in the proper order.

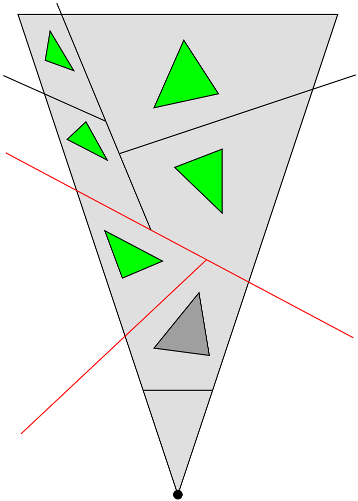


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

Recur... and draw the triangles in the proper order.

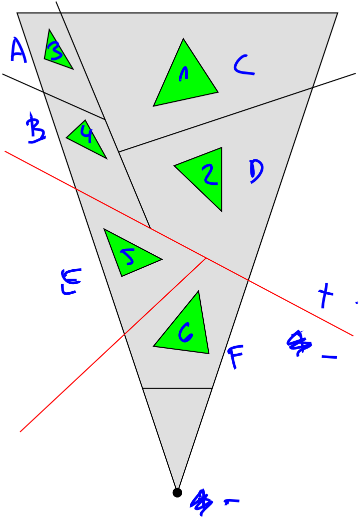


BSP-trees: basic idea

Goal: determine a drawing order for n triangles.

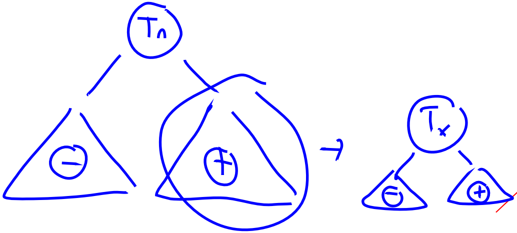
If we could find a plane that splits the set into two sets, then we could first draw the "far triangles", and then the "near triangles".

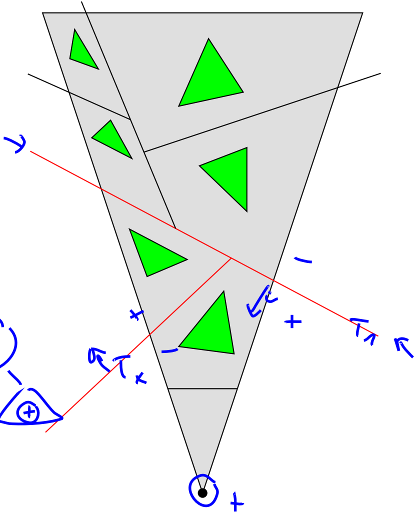
Recur. . . and draw the triangles in the proper order.



BSP-trees: data structure

Data structure to store the order information: Binary Space Partitioning tree BSP



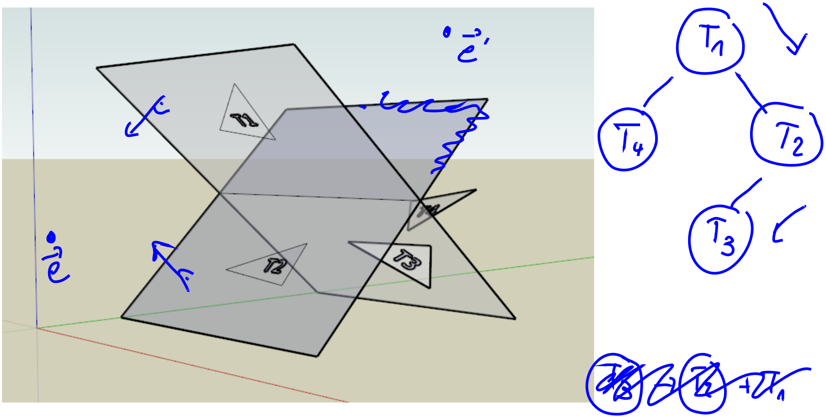


Drawing the triangles - algorithm

function `draw(bsptree tree, point e)`
if `(tree.empty)` **then**
  `return`
if $(f_{tree.root}(\mathtt{e}) < 0)$ **then**
  `draw(tree.plus, e)`
  `rasterize tree.triangle`
  `draw(tree.minus, e)`
else
  `draw(tree.minus, e)`
  `rasterize tree.triangle`
  `draw(tree.plus, e)`

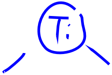
Drawing the triangles - example

(cf. book, fig. 8.4)

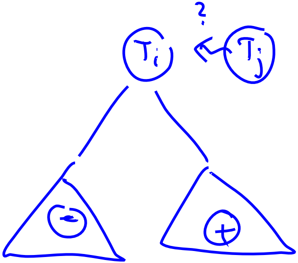


$T_4 - T_1 - T_3 - T_2$

Building the tree - algorithm (1)



tree-root = node(T_1)
for $i \in \{2, \ldots, N\}$ **do**
  tree-root.add(T_i)



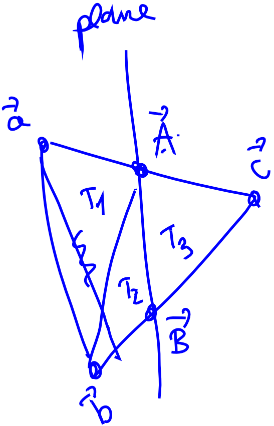
Building the tree - algorithm (2)

f ≙ plane defined by T root

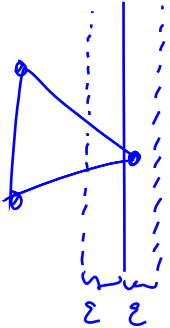
```
function add(triangle T)
if (f(a) < 0 and f(b) < 0 and f(c) < 0) then
  if (negative subtree is empty) then
    negative-subtree = node(T)
  else
    negative-subtree.add(T)
elsif (f(a) > 0 and f(b) > 0 and f(c) > 0) then
  if (positive subtree is empty) then
    positive-subtree = node(T)
  else
    positive-subtree.add(T)
else
  (we excluded this case in the assumptions)
```

Building the tree - special cases

(cf. book, fig. 8.5 and 8.6)



Precision problem if vertex is very close to splitting plane



if $-\varepsilon \leq f(\vec{c}) \leq \varepsilon$ then $f(\vec{c}) \overset{!}{=} 0$

Building the tree - algorithm (2^*)

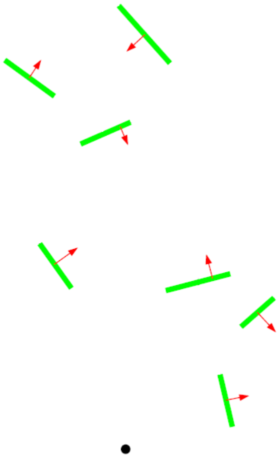
function `add(triangle` T`)`
if $(abs(f(\mathbf{a})) < \epsilon)$ **then** `fa = 0` **else** `fa =` $f(\mathbf{a})$
if $(abs(f(\mathbf{b})) < \epsilon)$ **then** `fb = 0` **else** `fb =` $f(\mathbf{b})$
if $(abs(f(\mathbf{c})) < \epsilon)$ **then** `fc = 0` **else** `fc =` $f(\mathbf{c})$
if $(fa \leq 0$ and $fb \leq 0$ and $fc \leq 0)$ **then**
  if `(negative subtree is empty)` **then**
    `negative-subtree = node(`T`)`
  else
    `negative-subtree.add(`T`)`
elsif $(fa \geq 0$ and $fb \geq 0$ and $fc \geq 0)$ **then**
  if `(positive subtree is empty)` **then**
    `positive-subtree = node(`T`)`
  else
    `positive-subtree.add(`T`)`
else *cut triangle into three triangles and add to each side*

Building the tree - summary

A standard way of building a BSP tree is:

- pick an arbitrary triangle, and make its supporting plane V the root of the tree.
- Any triangles that are intersected by the plane are split into three triangles that each lie on one side of the plane.
- Recur on the set of triangles in V^+, and make the resulting tree the left child of the root.
- Recur on the set of triangles in V^-, and make the resulting tree the right child of the root.

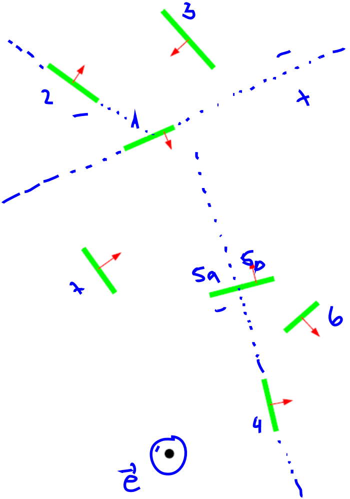
Traversing the tree

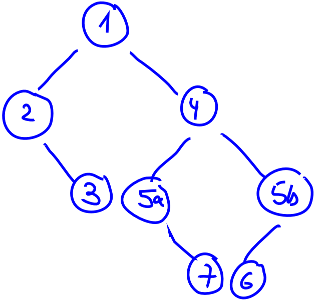


Test the viewing point against the plane of a node. If it is on the positive side, then first draw the subtree at the negative side, then the triangle stored in the node, and finaly the subtree on the positive side.

. . . otherwise, draw in opposite order.

Traversing the tree





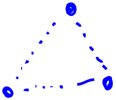
Drawing order for $\vec{e}$:

3 - 2 - 1 - 5b - 6 - 4 - 7 - 5a

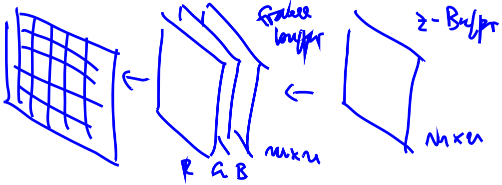
Z-buffer algorithm: basic idea

Apart from the frame buffer, which contains the pixels of the image, also maintain a Z-buffer of the same width and height, to store depth information for the projected triangles.

- Initialize all Z-buffer entries to $z_{\max}$. (far plane f)
- If a pixel is to be drawn at position $[i, j]$, first test if its corresponding z-value p_z is smaller then Z-buffer$[i, j]$.
- If this is the case, draw the pixel, and update Z-buffer$[i, j]$ to p_z.
- Otherwise, do not draw the pixel.
- z-values for projected vertices are calculated; for remaining pixels, they are interpolated.

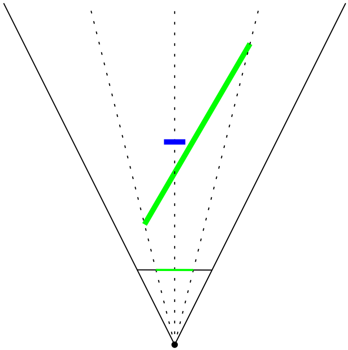


m×n



Interpolating z-values

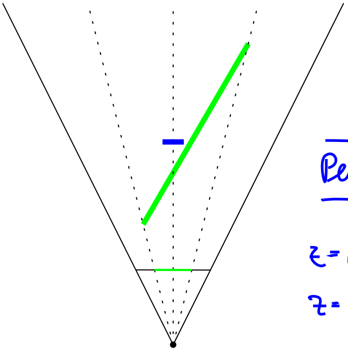
scanline conversion



If we do perspective transformation, then we must be careful with interpolating z-values.

Fortunately, these problems disappear if we transform to the orthographic or canonical view volume (review the requirements on the perspective projection matrix that we formulated).

Interpolating z-values



$\boxed{n \le z \le f}$ $\quad z \in \{0, 1, \dots, B-1\}$

b = # bits needed: $B = 2^b$

Resolution: $\Delta z = \frac{f-n}{B} = \frac{f-n}{2^b}$

Persp. proj.: $n < z < f$

$z' = n + f - \frac{f \cdot n}{z}$

$z = n \leadsto z' = n$

$z = f \leadsto z' = f$

$z' \sim \frac{1}{z}$