Computer Graphics

Hidden surface removal

Konstantin Tretyakov

kt@ut.ee

Oct 09, 2013

Standard Graphics Pipeline

- Vertex transform
  - $x' = PVMx$
  - **Perspective division**
  - **Viewport transform**
- Culling and clipping
- Rasterization
- Fragment shading
- Visibility tests & blending

Standard Graphics Pipeline

- Vertex transform
  - $x' = PVMx$
  - **Perspective division**
  - **Viewport transform**
- Culling and clipping
- Rasterization
  - **Triangles, lines, etc**
  - **Attribute interpolation**
- Fragment shading
- Visibility tests & blending

Standard Graphics Pipeline

Vertex transform

- $x' = PVMx$
- **Perspective division**
- **Viewport transform**

Culling and clipping

Rasterization

- **Triangles, lines, etc**
- **Attribute interpolation**

Fragment shading

Visibility tests & blending

Today

Hidden surface removal

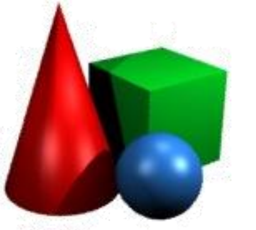
- **Why bother?**

Hidden surface removal

- **Why bother?**
  - Ensure the polygons occlude each other as necessary.
  - Maximize performance.

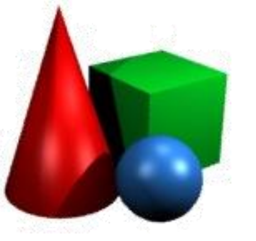
Hidden Surface Removal

- **Sorting** (polygons)
- **Visibility tests** (pixels)
- **Culling** (objects/polygons)
- **Clipping** (polygons/lines)



Hidden Surface Removal

- **Sorting** (polygons)
- **Visibility tests** (pixels)
- **Culling** (objects/polygons)
- **Clipping** (polygons/lines)



Polygon sorting

- “Painter’s algorithm” – the most obvious solution to ensure proper polygon occlusion: draw polygons back-to-front.

Polygon sorting

- “Painter’s algorithm” – the most obvious solution to ensure proper polygon occlusion: draw polygons back-to-front.

Does it always guarantee correct rendering?

Polygon sorting

- "Painter's algorithm" – the most obvious solution to ensure proper polygon occlusion: draw polygons back-to-front.

Does it always guarantee correct rendering?

- No, but in many cases the models are designed so that the counterexample is not possible.

Algorithmics quiz

- We would have to sort triangles every time viewpoint changes. This is inefficient.
- How to avoid the need to sort triangles every time?

Algorithmics quiz

- We would have to sort triangles every time viewpoint changes. This is inefficient.

- How to avoid the need to sort triangles every time?

  - Hint:

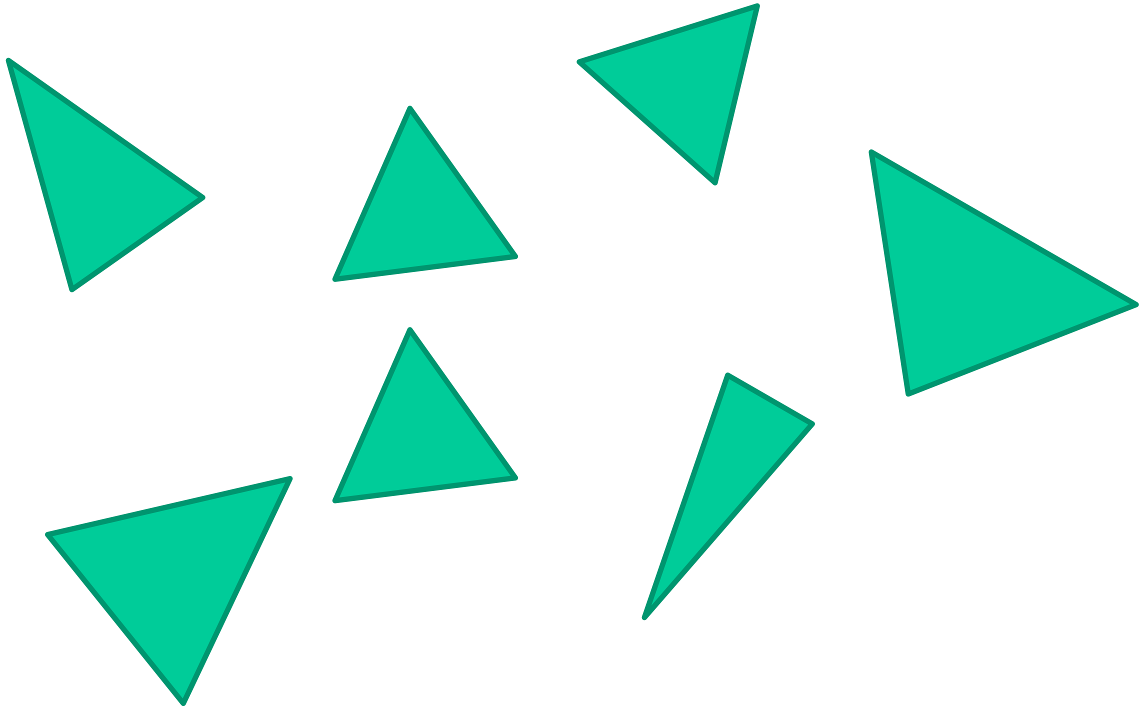
Algorithmics quiz

- We would have to sort triangles every time viewpoint changes. This is inefficient.

- How to avoid the need to sort triangles every time?

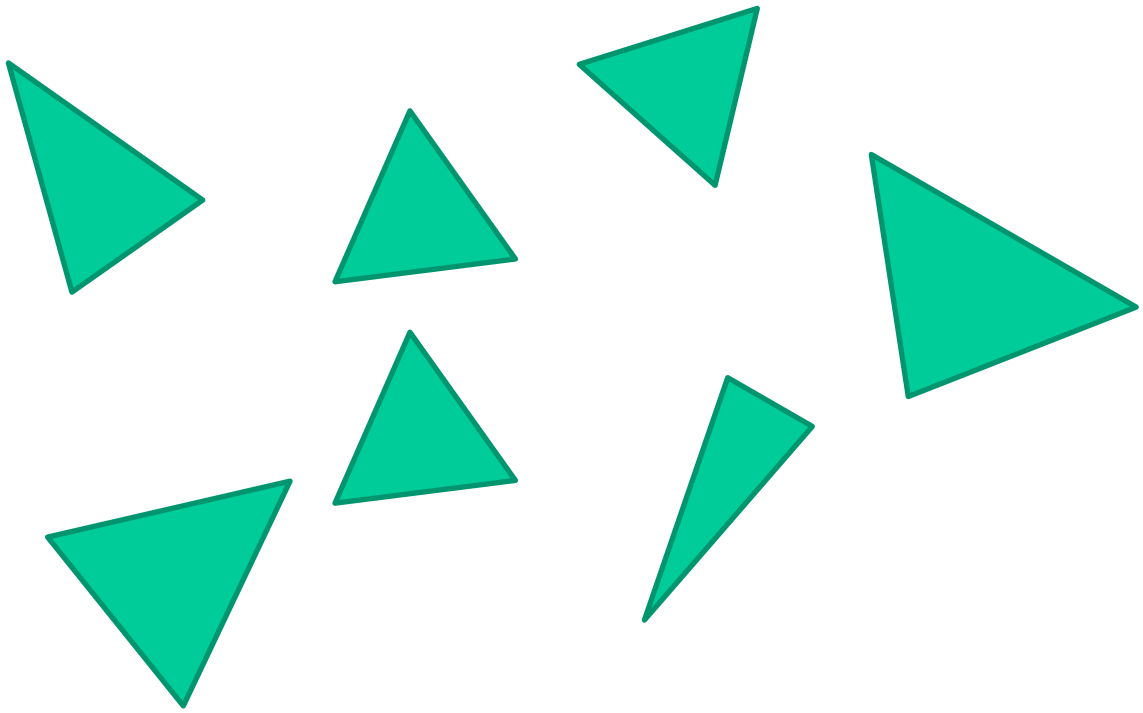
Binary space partitioning (BSP)-trees

BSP-trees



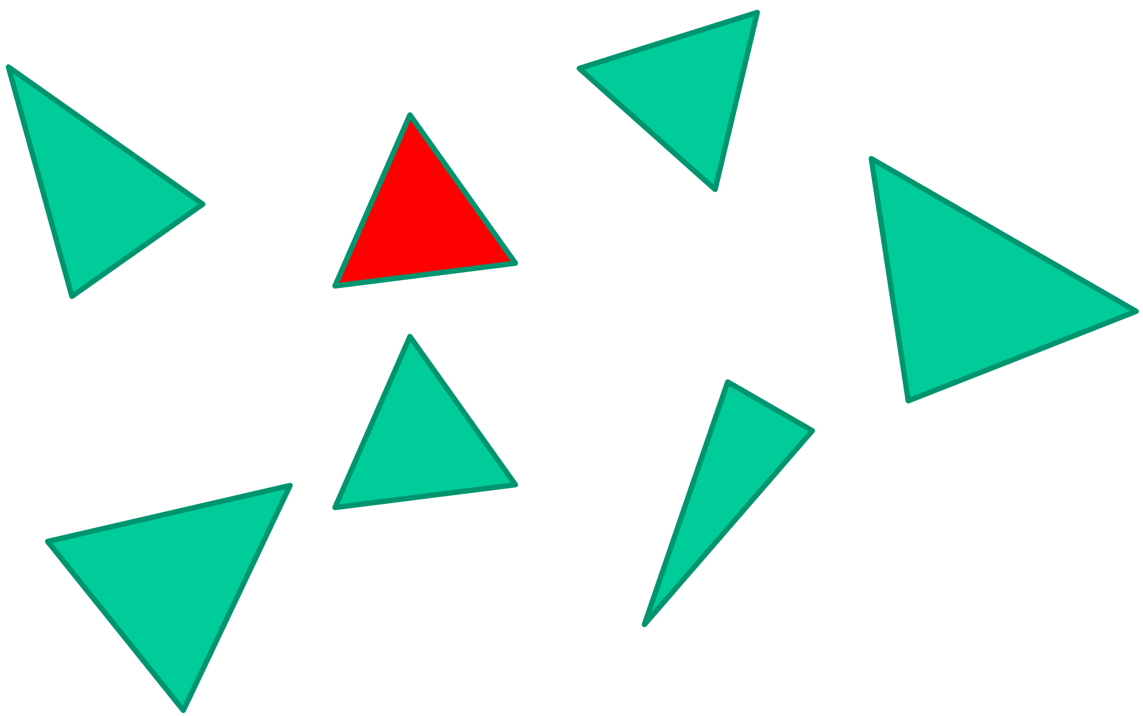
BSP-trees

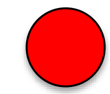
Start with the set of all polygons in the scene



BSP-trees

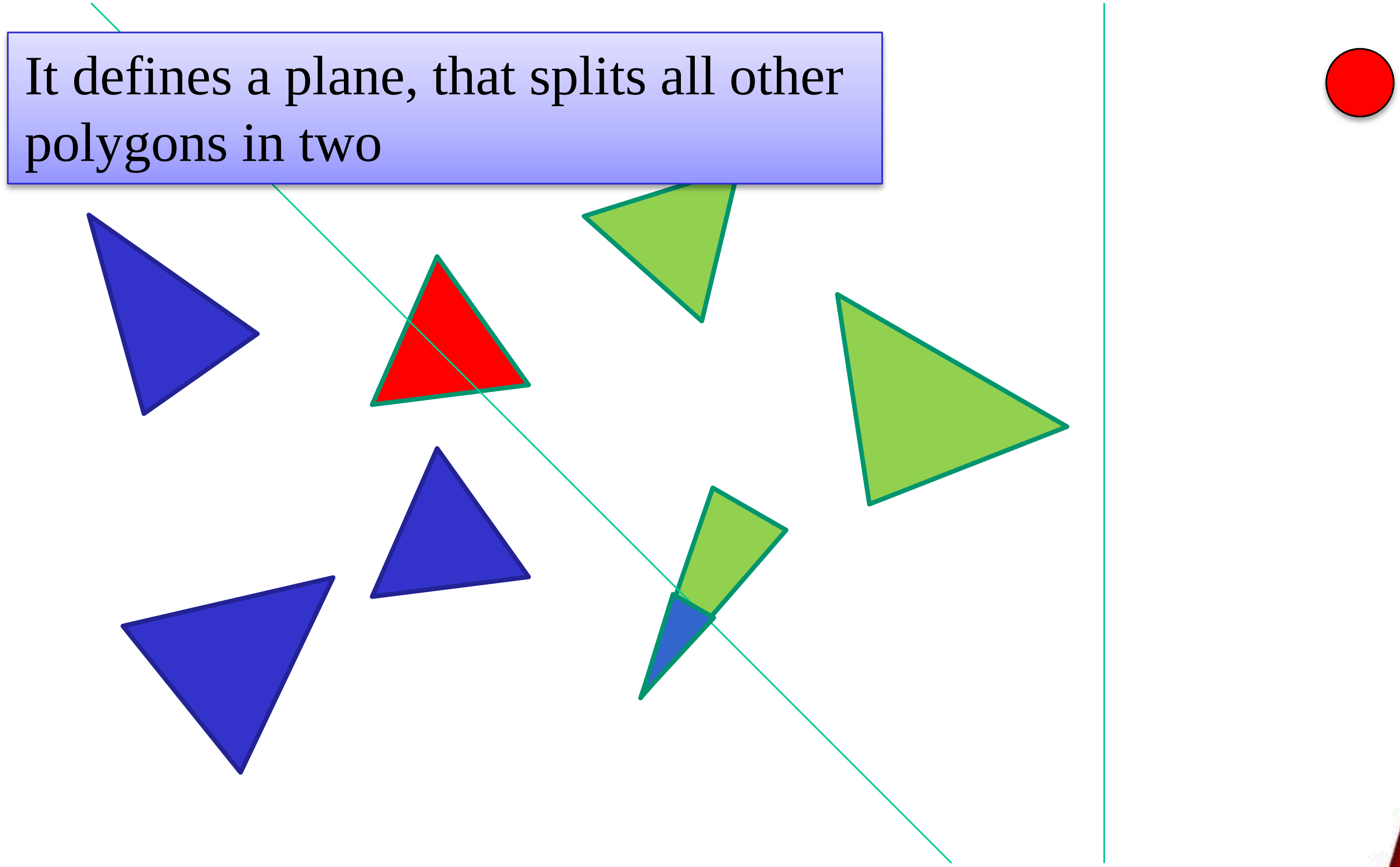
Pick one as the root





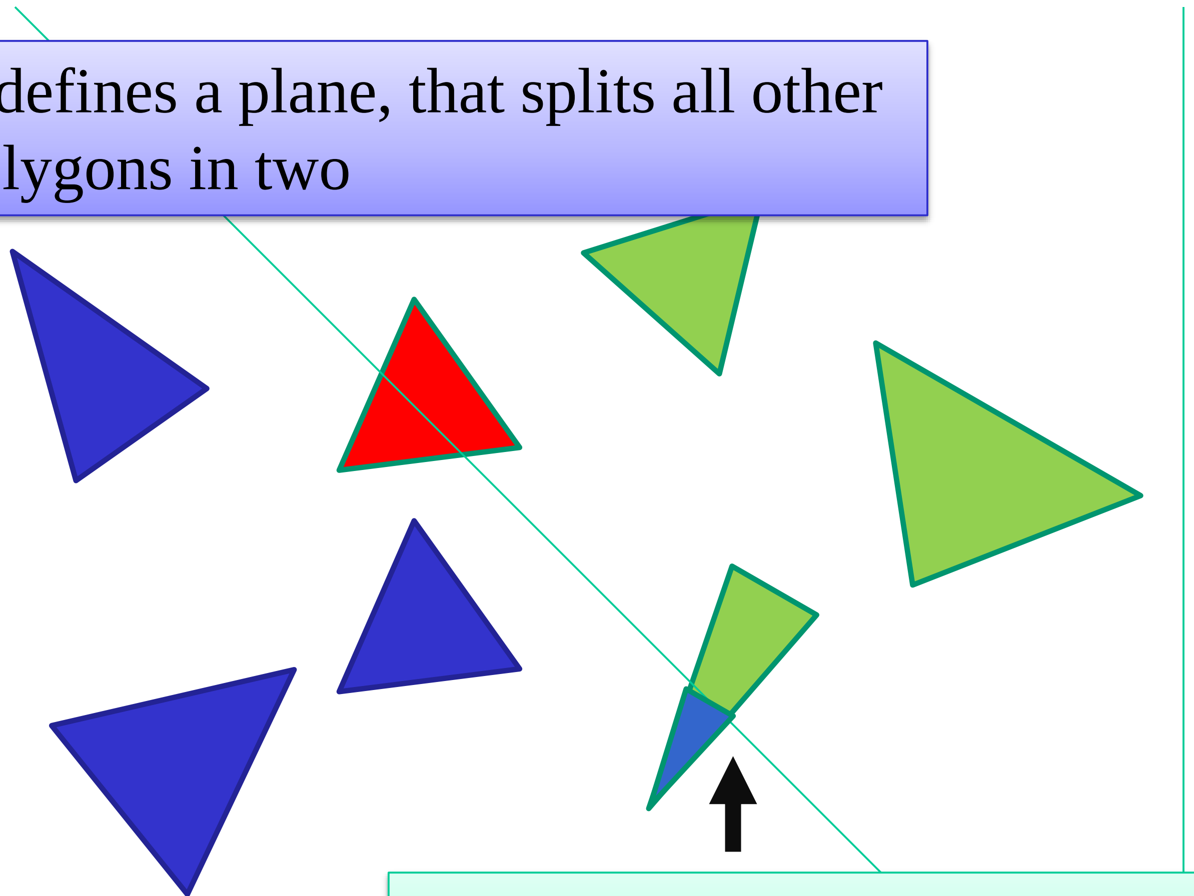
BSP-trees

It defines a plane, that splits all other polygons in two



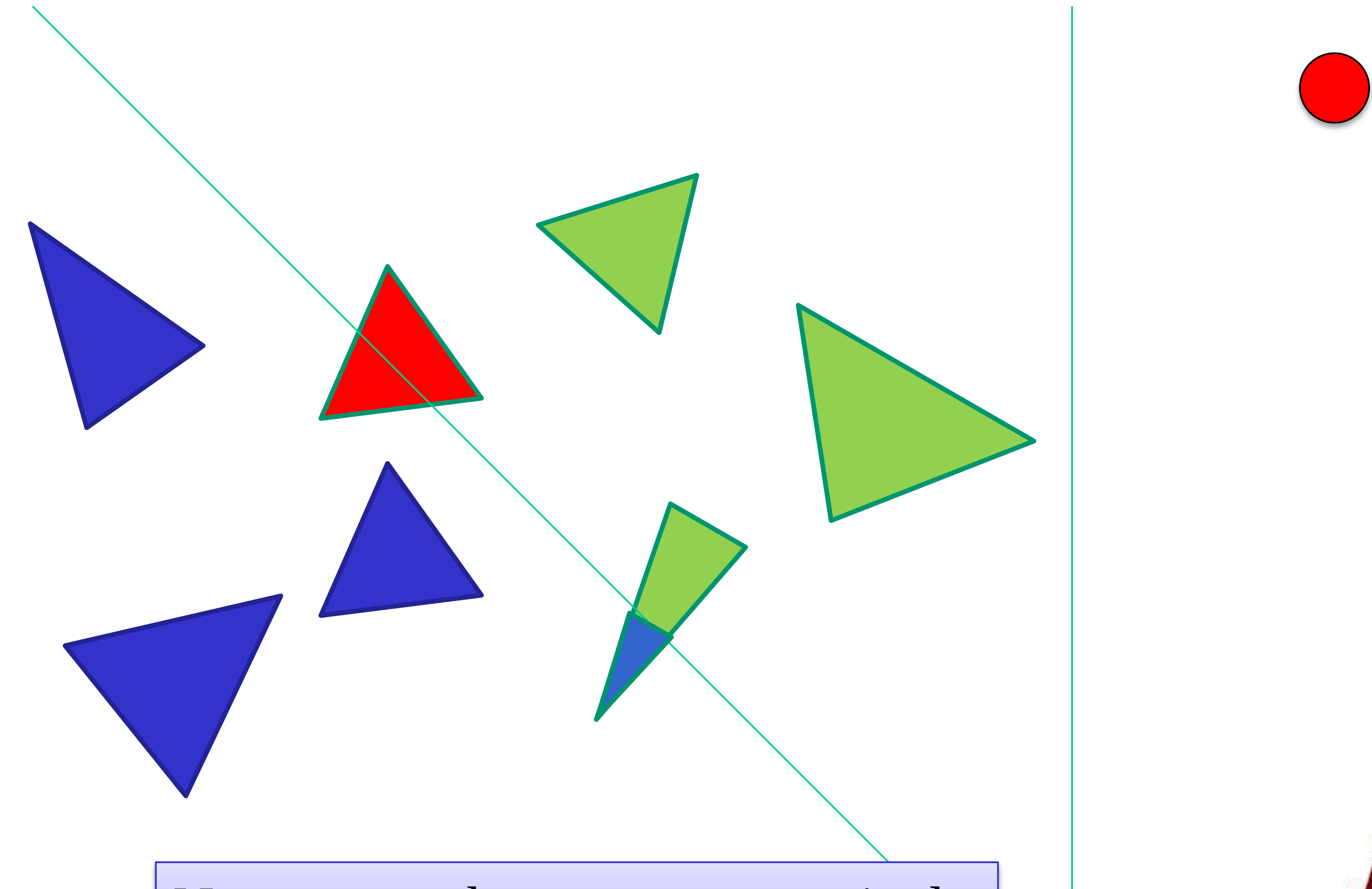
BSP-trees

It defines a plane, that splits all other polygons in two



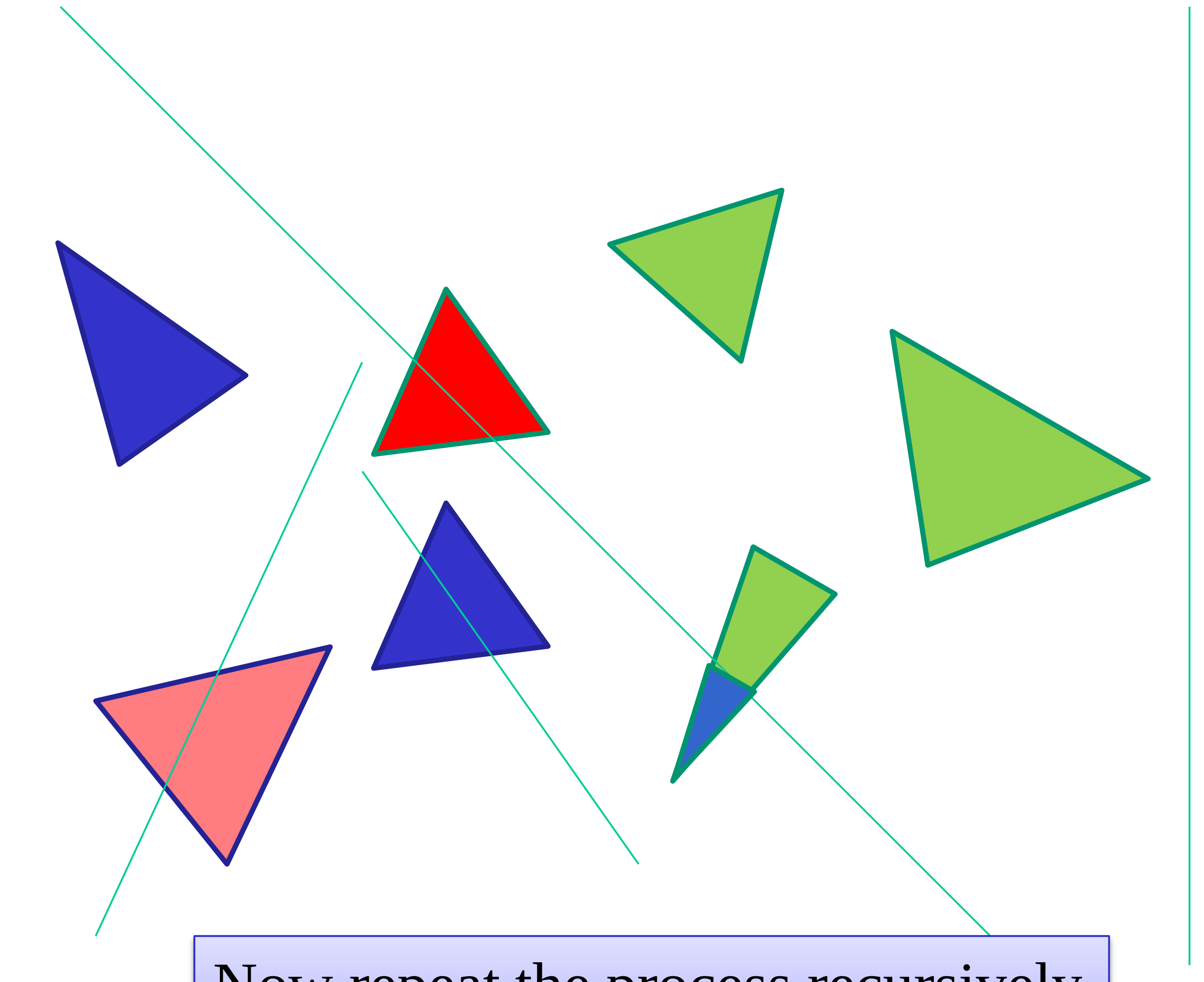
Some polygons will have to be split in two separate polygons

BSP-trees



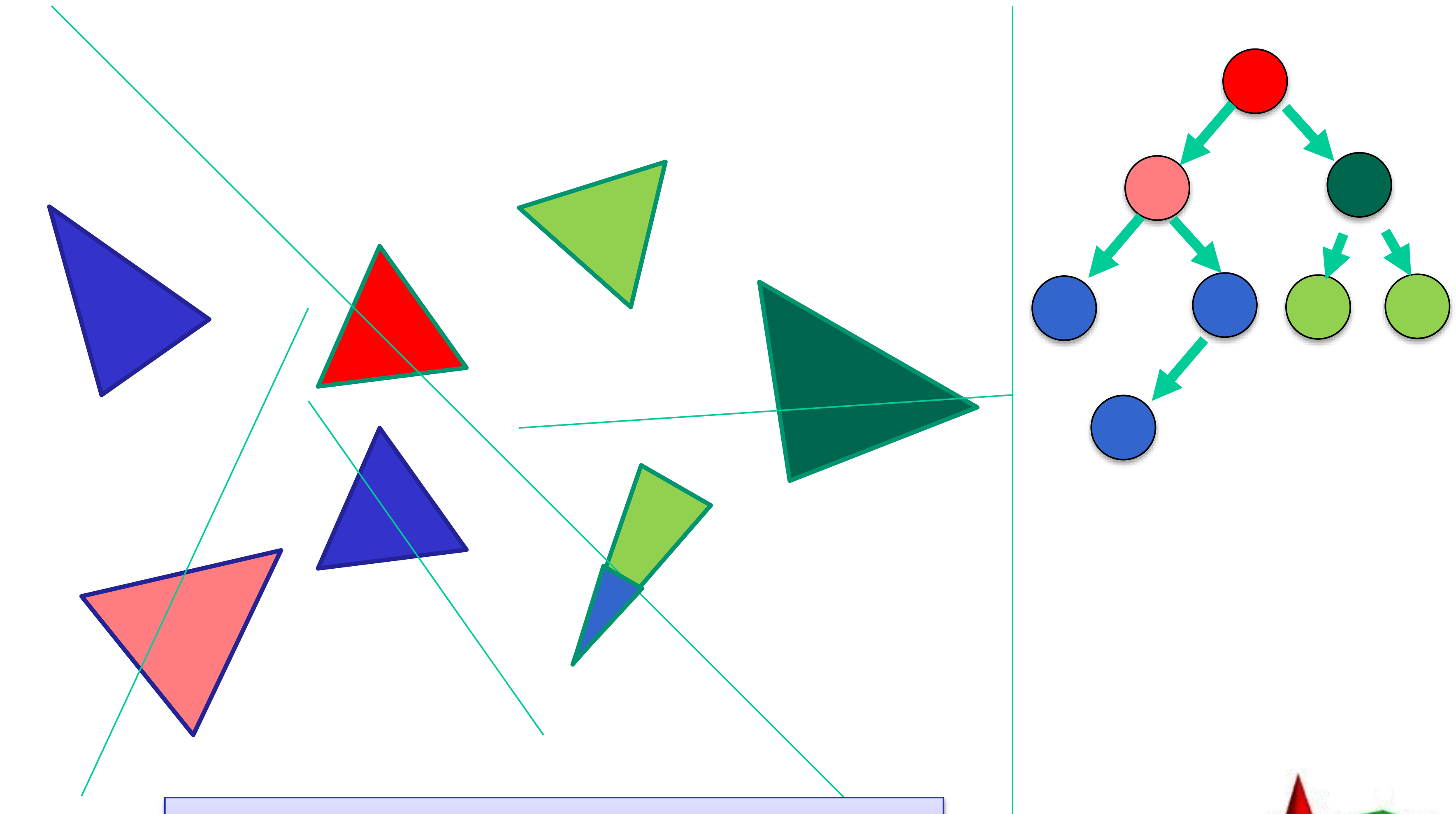
Now repeat the process recursively for each subspace

BSP-trees



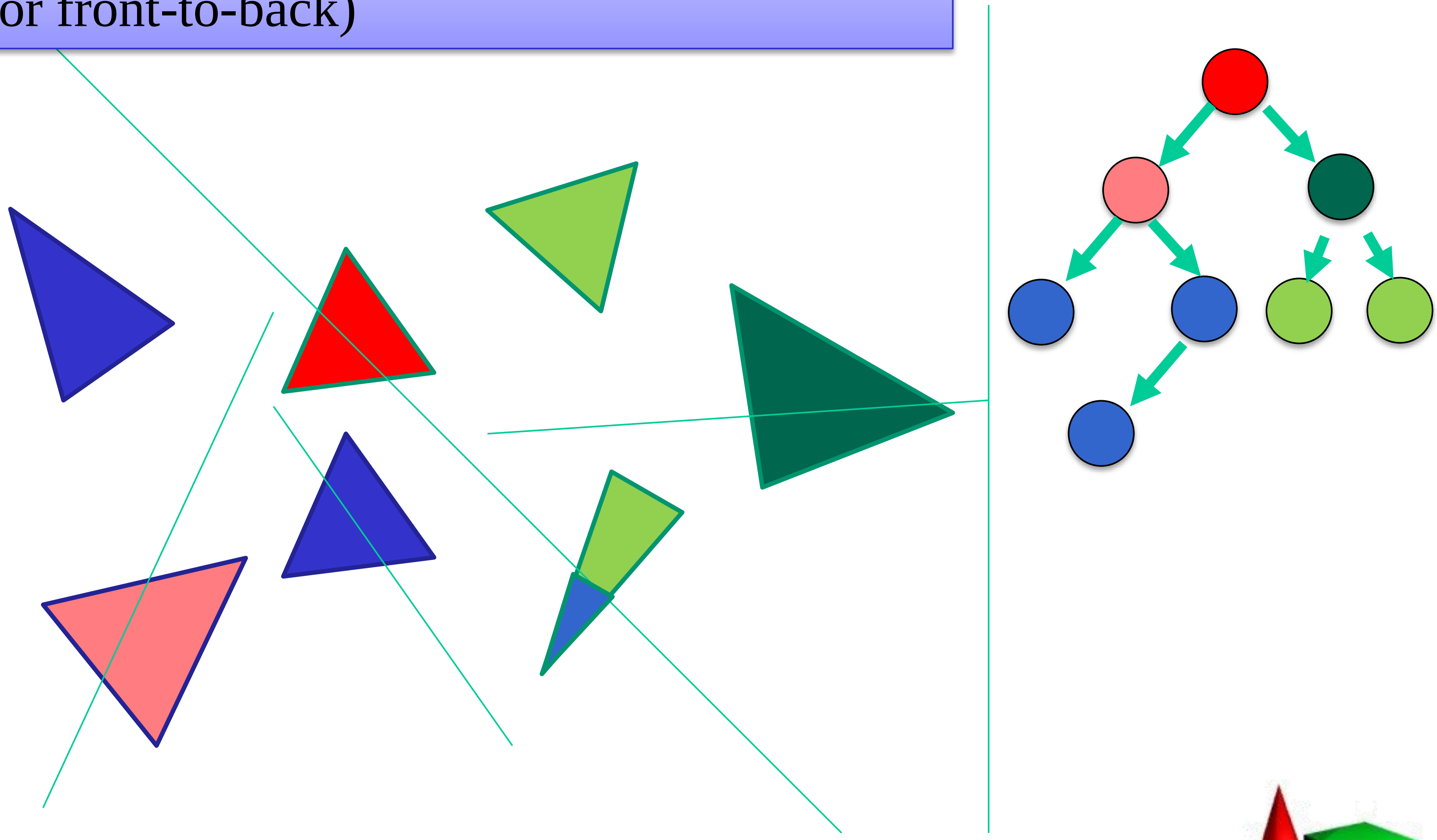
Now repeat the process recursively for each subspace

BSP-trees



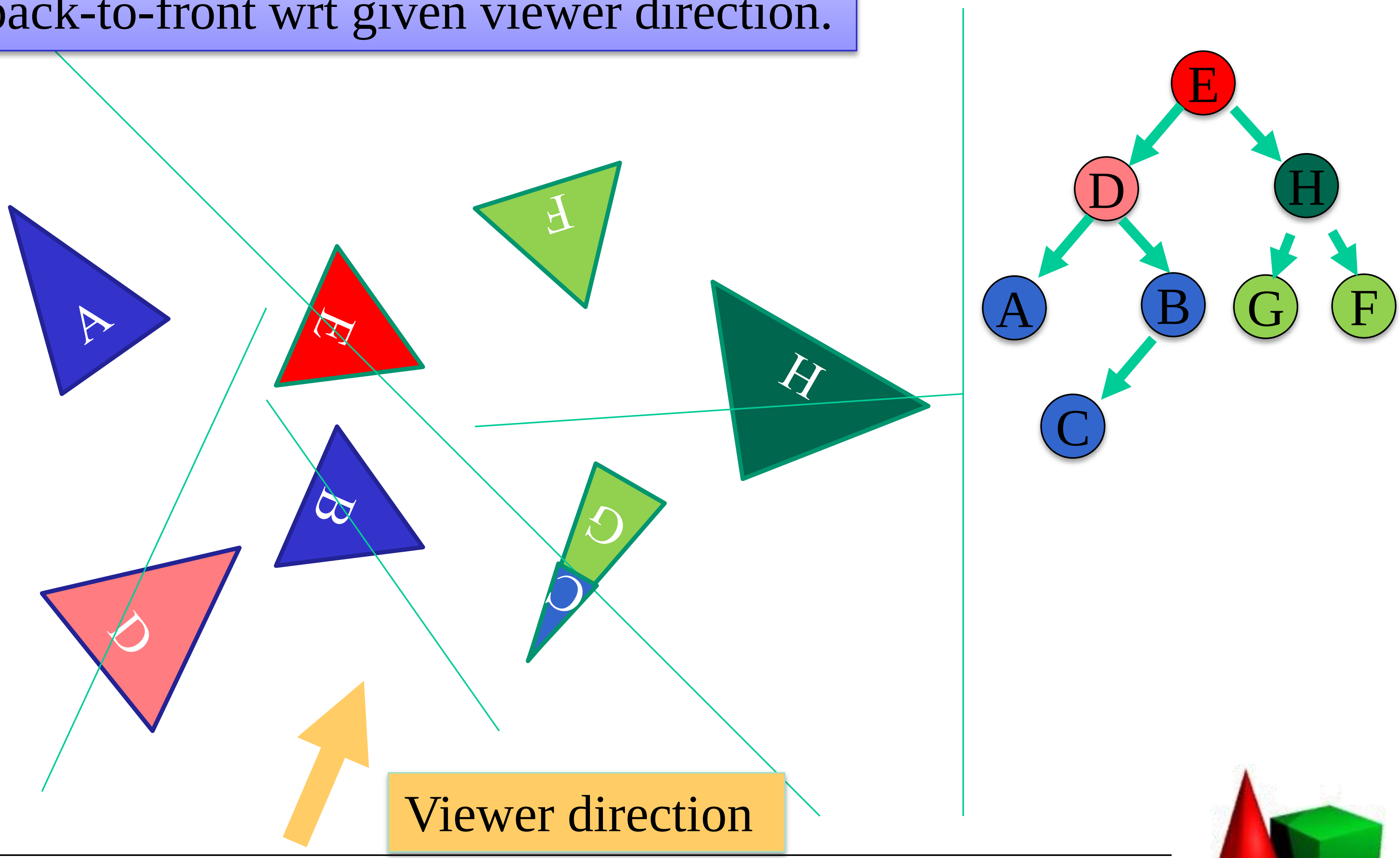
Now repeat the process recursively for each subspace

Now for any orientation of the scene we can quickly enumerate polygons back-to-front (or front-to-back)



Quiz

Use the BSP-tree to enumerate polygons back-to-front wrt given viewer direction.



Hidden Surface Removal

- **Sorting** (polygons)
- **Visibility tests** (pixels)
- **Culling** (objects/polygons)
- **Clipping** (polygons/lines)

Hidden Surface Removal

- **Sorting** (polygons)
- **Visibility tests** (pixels)
- **Culling** (objects/polygons)
- **Clipping** (polygons/lines)

Z-buffer

- For each rendered pixel, write a depth value to a separate buffer.
- Discard pixels for which there is already a smaller value written in the depth buffer.

Z-buffer

- When using Z-buffer, does the order of polygon rendering matter?

Z-buffer

- When using Z-buffer, does the order of polygon rendering matter?
  - Visually: **no**. (unless you have transparency)
  - Efficiency-wise: **yes**.
    - It is more efficient to render **front-to-back**. BSP trees can be used again.
    - To avoid complicated per-pixel computations it is sometimes best to render the whole scene to the Z-buffer first without any per-pixel effects, and then render **again** as normal without clearing the buffer relying on the *early depth test*.

Z-buffer in OpenGL

- Z-buffer is supported in hardware

```
glutInitDisplayMode(GLUT_RGBA |
                    GLUT_DOUBLE |
                    GLUT_DEPTH)
```

```
glDepthMask(GL_TRUE);
```

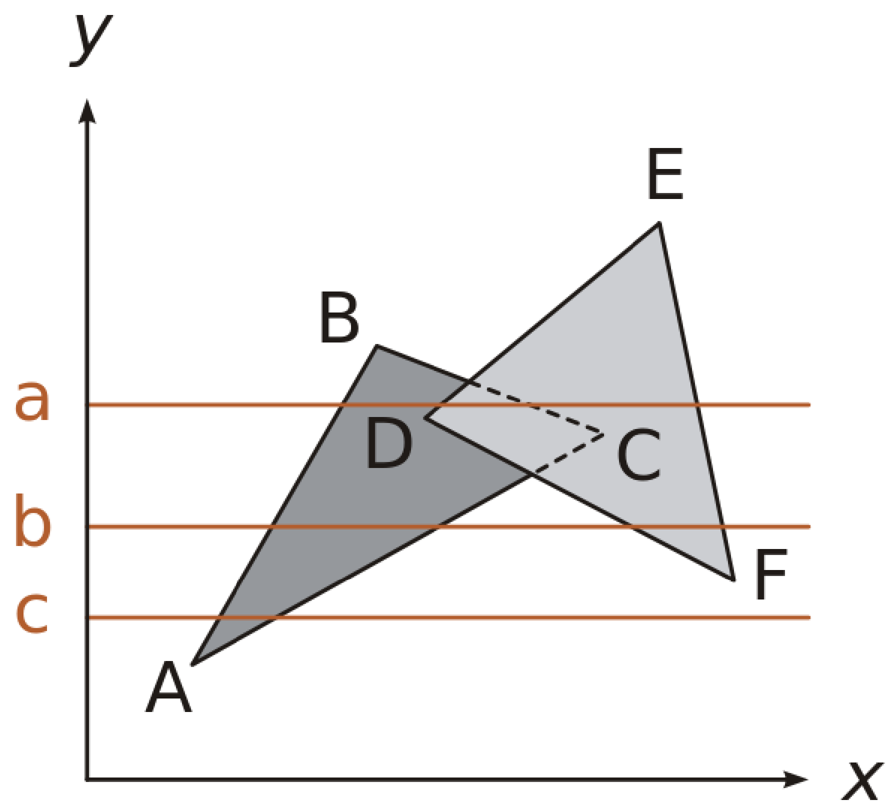
```
glEnable(GL_DEPTH_TEST);
```

```
glDepthFunc(GL_LESS);
```

```
glReadPixels(… GL_DEPTH_COMPONENT …);
```

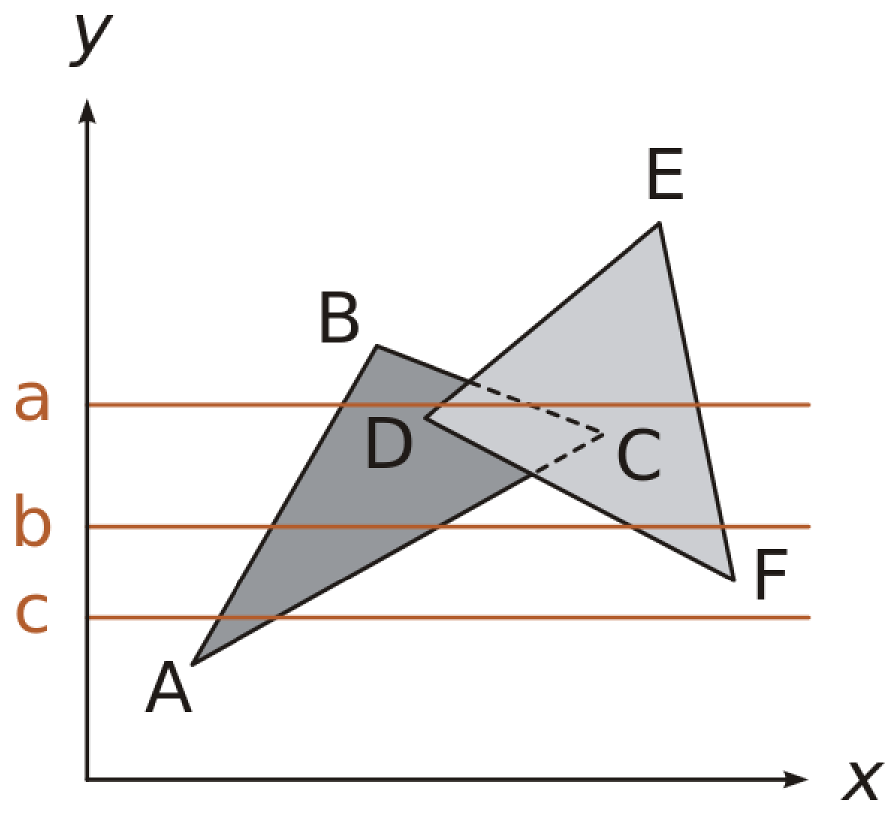
Scanline rendering

- Alternative method of rasterization which, like Z-buffer deals with depth on a per-pixel basis.



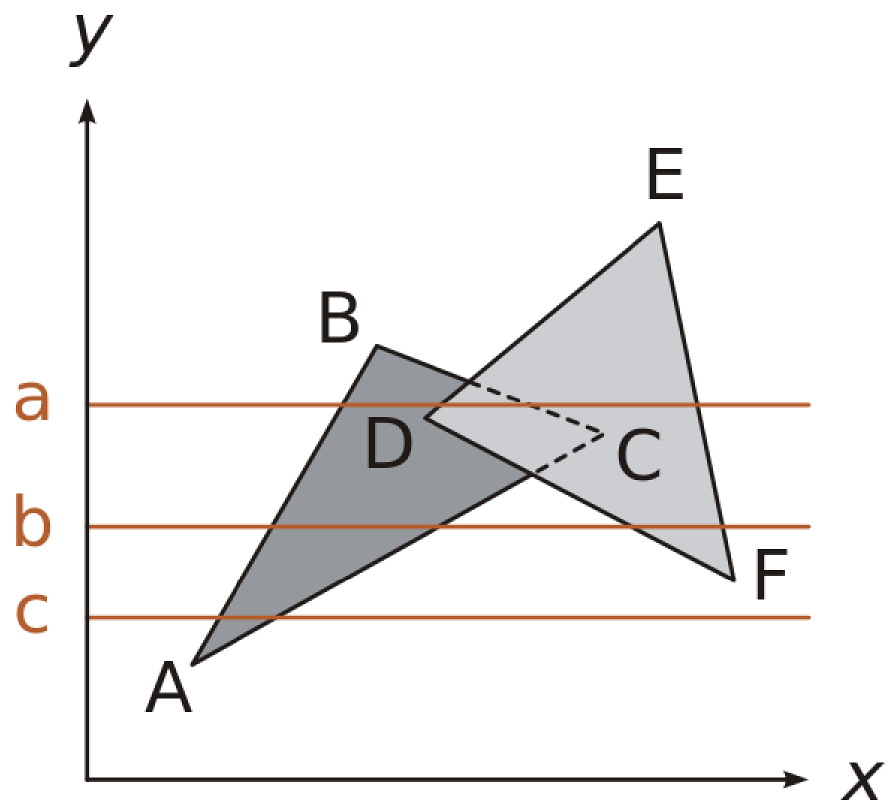
Scanline rendering

- Instead of rasterizing polygon-by-polygon, we can rasterize line-by-line, keeping track which polygons intersect the current line



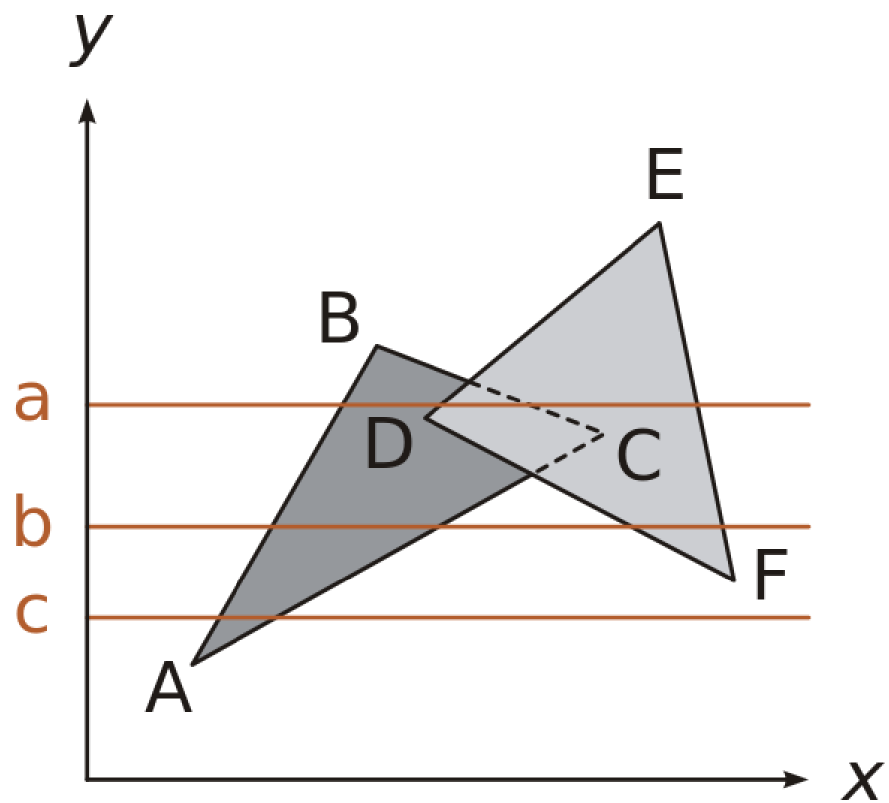
Scanline rendering

- The rendering can be faster than Z-buffer for high-depth scenes



Scanline rendering

- Quake used it.
- Nintendo DS (2004) presumably used it.
- Otherwise, it is of largely theoretical interest nowadays, as it does not fit the logic of modern GPUs.



Hidden Surface Removal

- **Sorting** (polygons)
- **Visibility tests** (pixels)
- **Culling** (objects/polygons)
- **Clipping** (polygons/lines)

Hidden Surface Removal

- **Sorting** (polygons)
- **Visibility tests** (pixels)
- **Culling** (objects/polygons)
- **Clipping** (polygons/lines)

Culling

- So far we have the following rendering algorithm in mind:
  - Enable Z-buffer
  - For each triangle in the scene
    - Rasterize (perhaps taking Z-buffer into account)

Culling

- So far we have the following rendering algorithm in mind:

What is still wrong here?

  - Enable Z-buffer
  - For each triangle in the scene
    - Rasterize (perhaps taking Z-buffer into account)

Culling

- So far we have the following rendering algorithm in mind:

What is still wrong here?

  - Enable Z-buffer
  - $\boxed{\text{For each triangle}}$ in the scene
    - Rasterize (perhaps taking Z-buffer into account)

Most triangles are usually not visible

Culling

- S
  a

t)

M

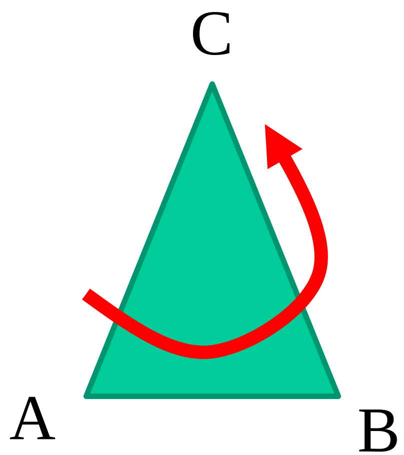
Culling

- Back-face culling
- Frustum culling
- Occlusion culling

Back face culling

- Each triangle has a "front" face and a "back" face, depending on the order of vertices.

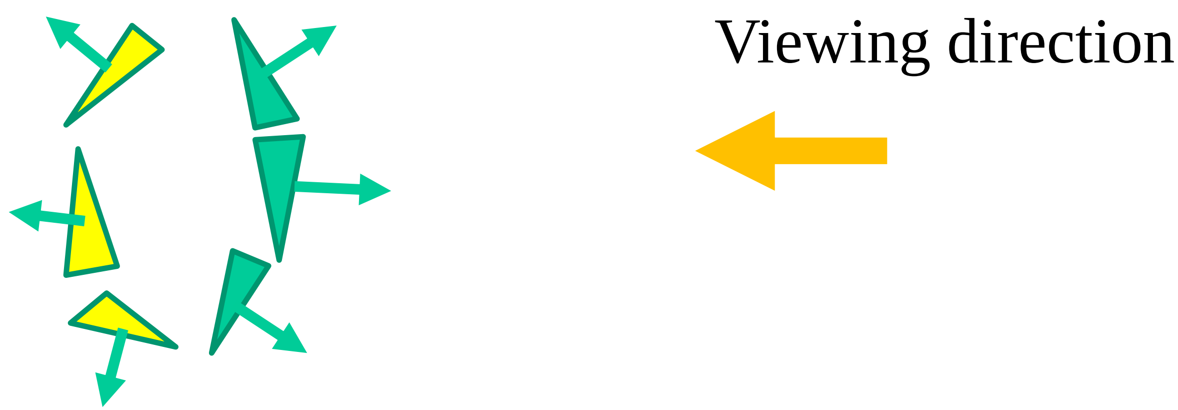
```
glFrontFace(GL_CCW);  // (default)
// Counter-clockwise defines front face
```



Back face culling

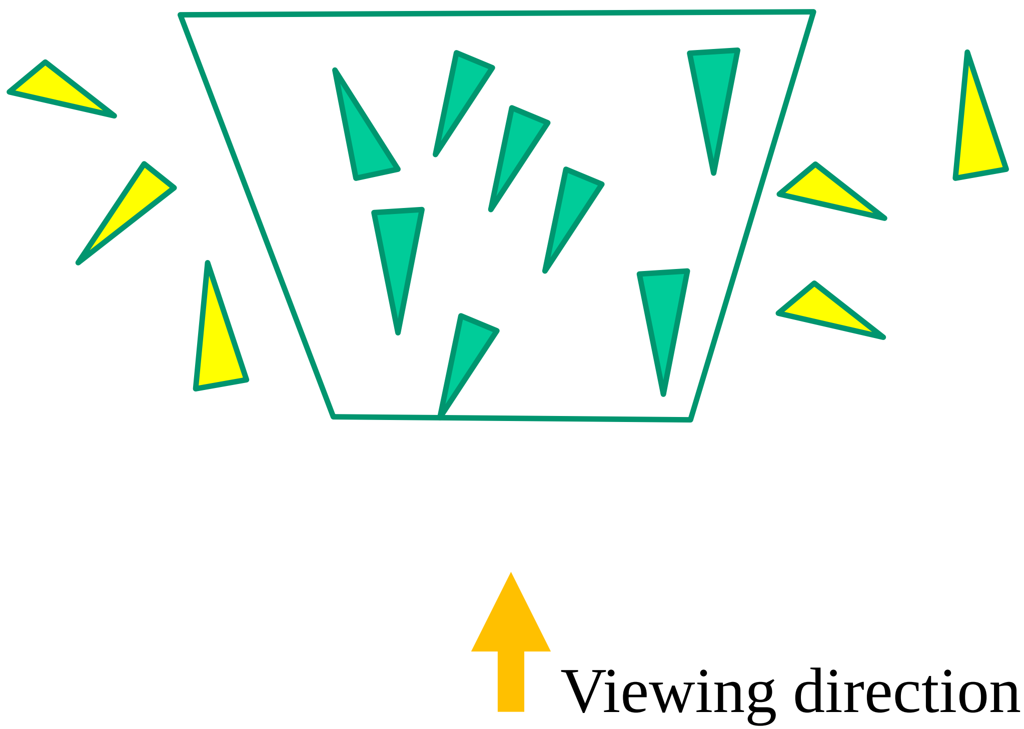
- Now we can determine for each triangle, whether its front face is looking towards the viewer. If not, we do not rasterize the triangle

```
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```



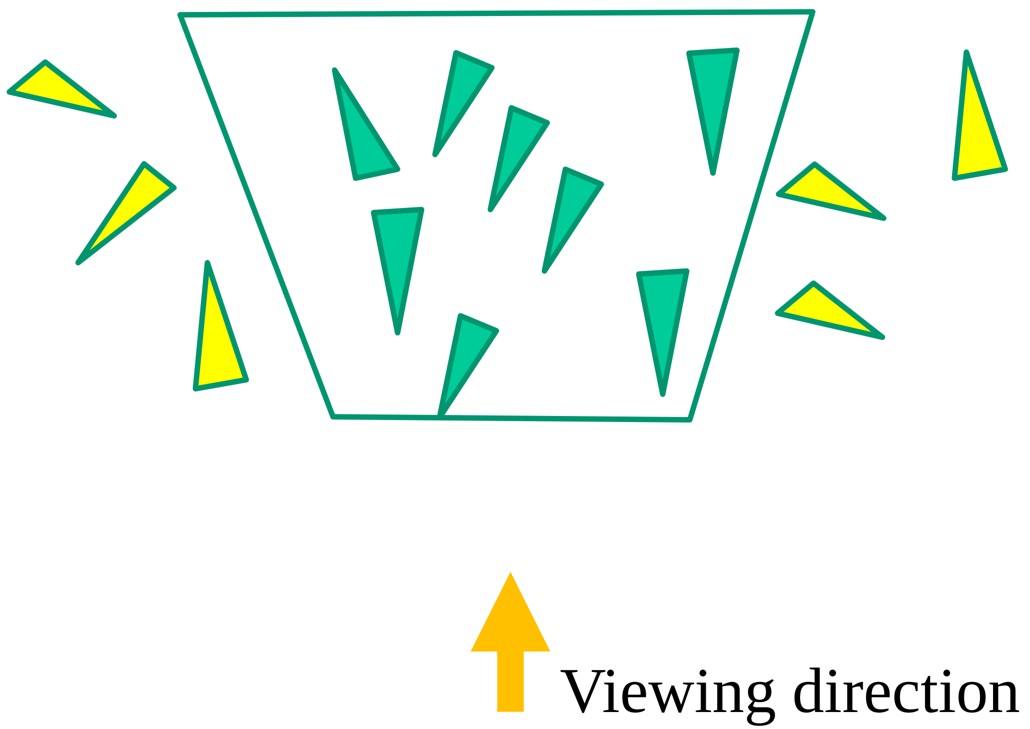
Frustum culling

- There is no point in trying to rasterize anything outside the view frustum:



Frustum culling

- You can cull polygons and objects BEFORE vertex shading and AFTER it.

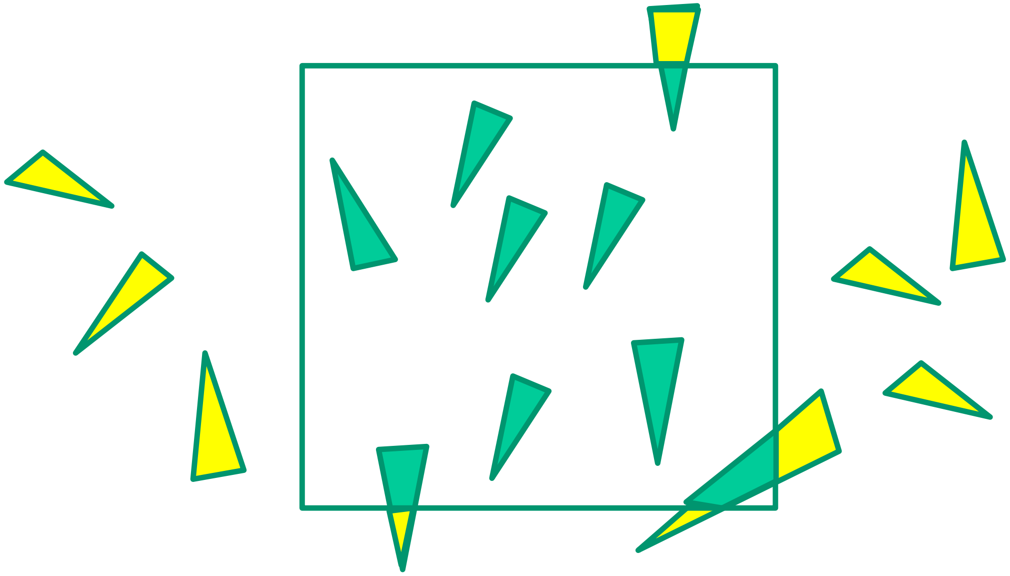


Frustum culling

- Culling AFTER the vertex shader:
  - Map all vertices through Model-View-Projection
  - Triangles that are outside of the clip space are ignored.
  - Triangles that intersect the clip space are *clipped*.

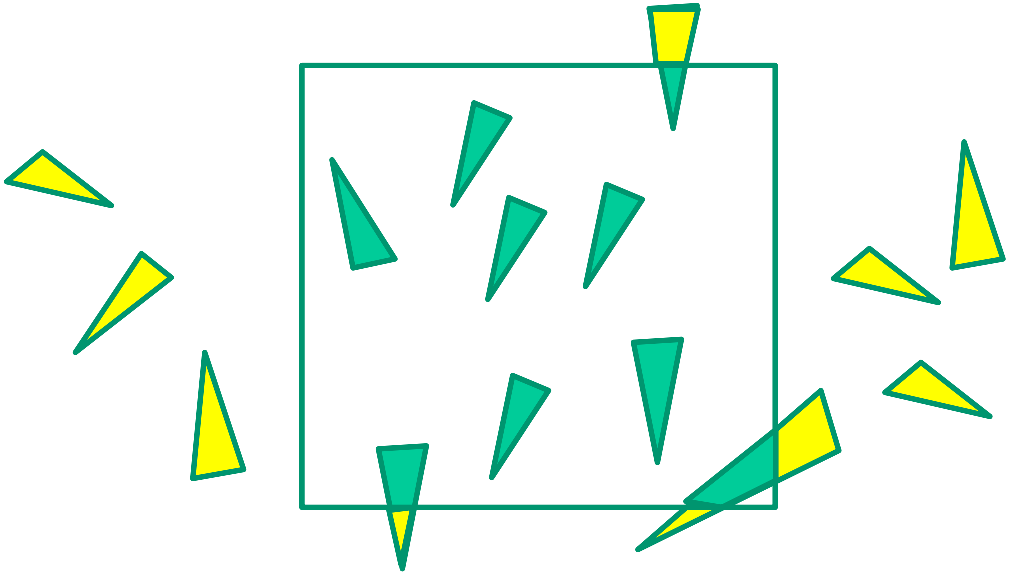
Frustum culling

- Culling AFTER the vertex shader:



Frustum culling

- Culling AFTER the vertex shader:



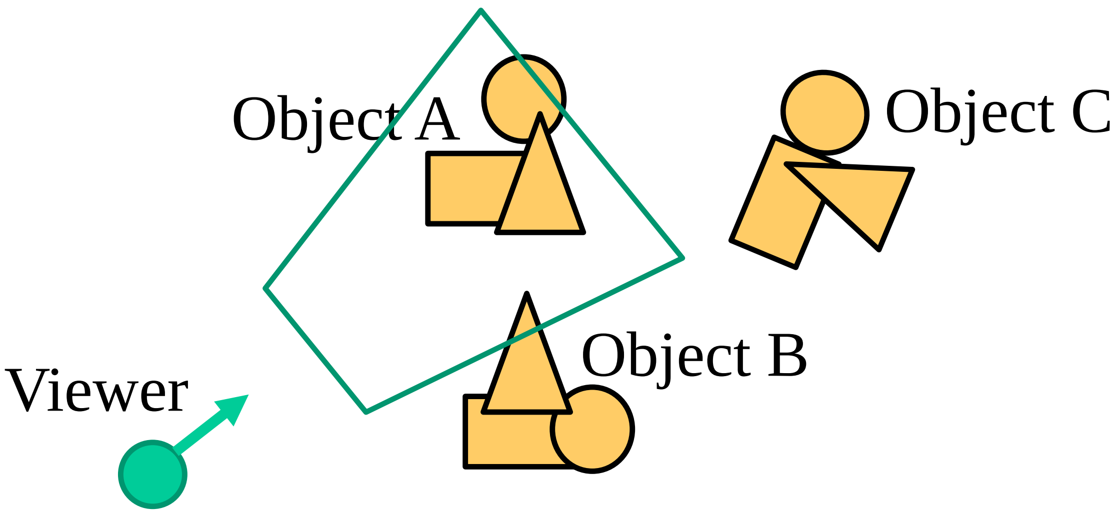
- Simple. Happens automatically in the GPU. Too late.

Frustum culling

- Culling BEFORE the vertex shader:
  - Slightly more complicated (you have to figure out what objects to draw before applying any model-view transforms, the cull volume is complicated)
  - Can prevent **huge** numbers of triangles from being unnecessarily sent to the GPU.

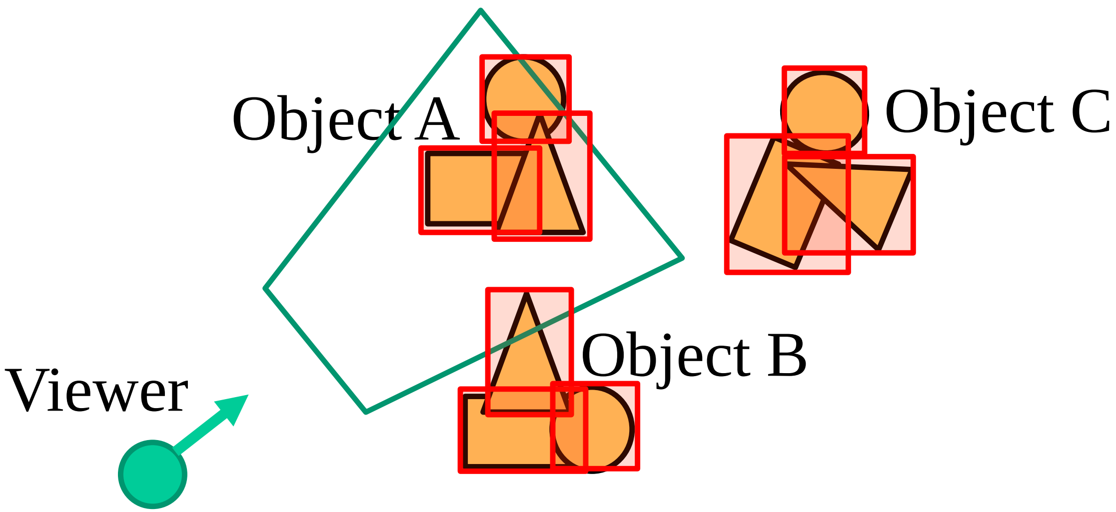
Frustum culling

- Given viewer’s position in the world, and the positions of objects, how to efficiently decide which objects can be culled?



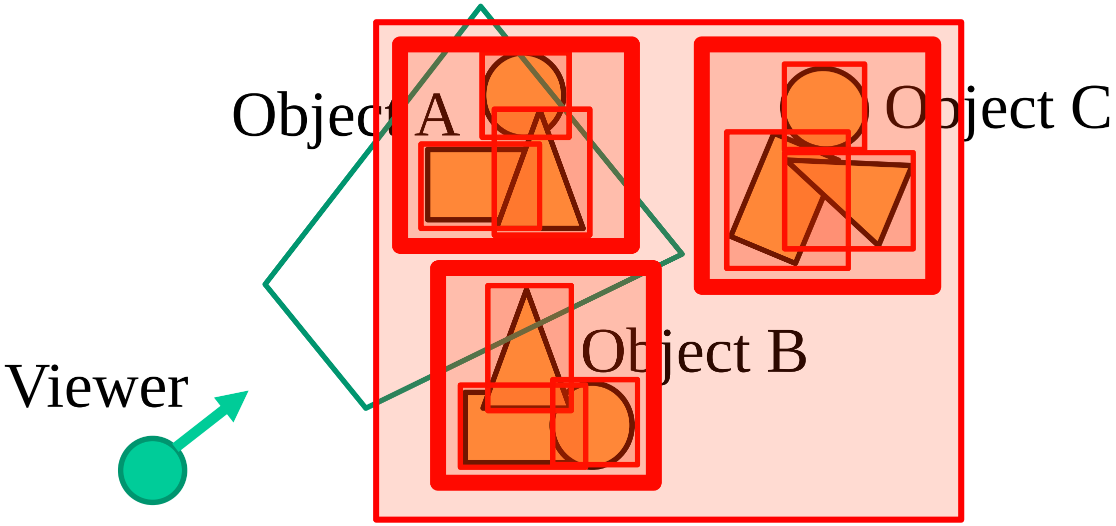
Frustum culling

- Idea #1: Replace objects with bounding boxes. Test those boxes for intersection with the frustum.



Frustum culling

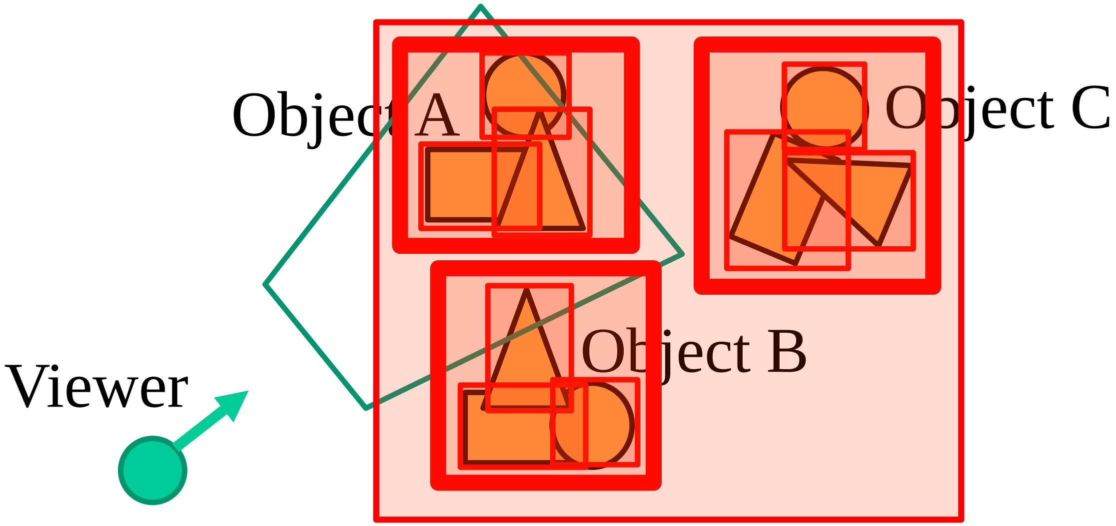
- Idea #2: Organize boxes into a hierarchy



Frustum culling

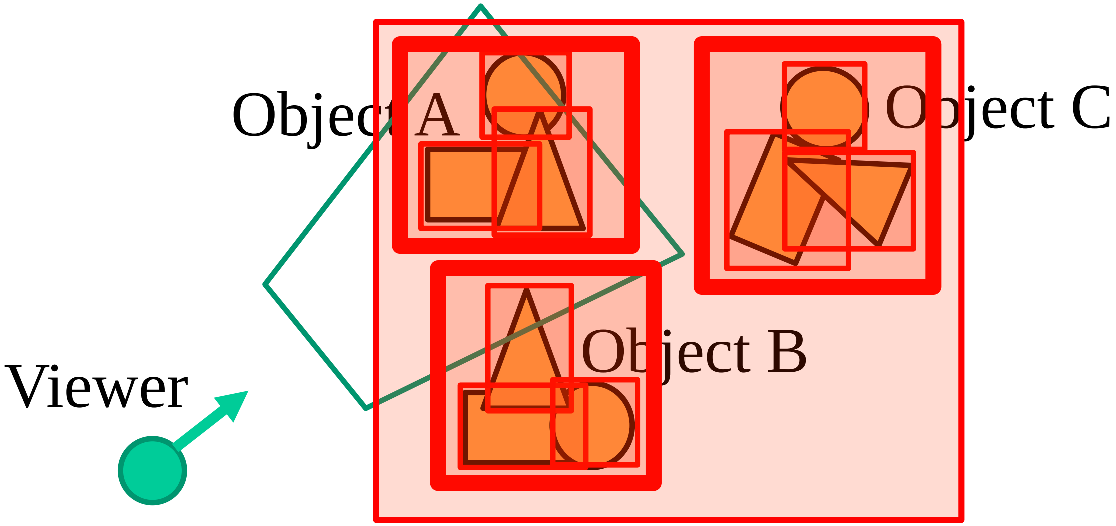
- Idea #2: Organize boxes into a hierarchy

Bounding volume hierarchy (BVH)
Axis-Aligned Bounding-Box Tree (AABB Tree)



Frustum culling

- Recursively test which boxes intersect the view volume. Discard everything in the boxes which fail the test

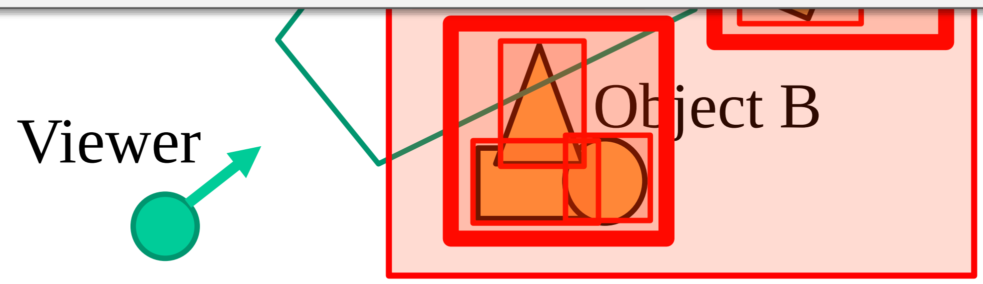


Frustum culling

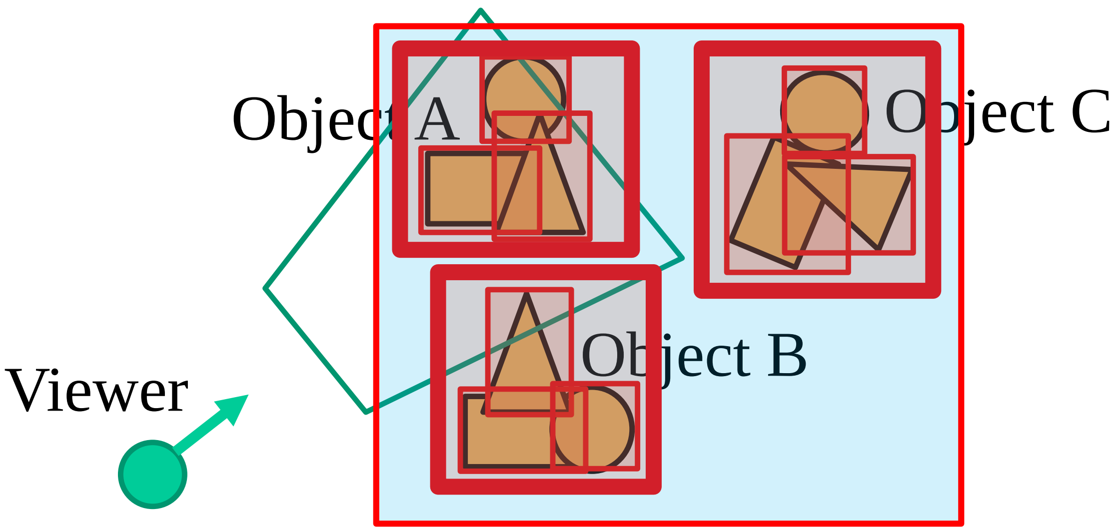
- Recursively test which boxes intersect the view volume. Discard everything in the

You may use OpenGL to test this:

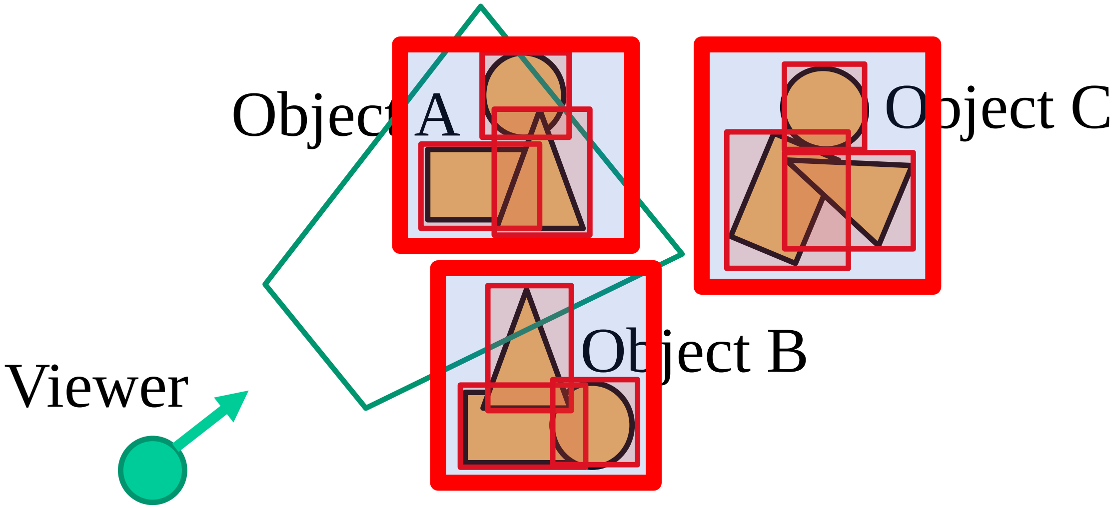
```
glRenderMode(GL_SELECT);
```



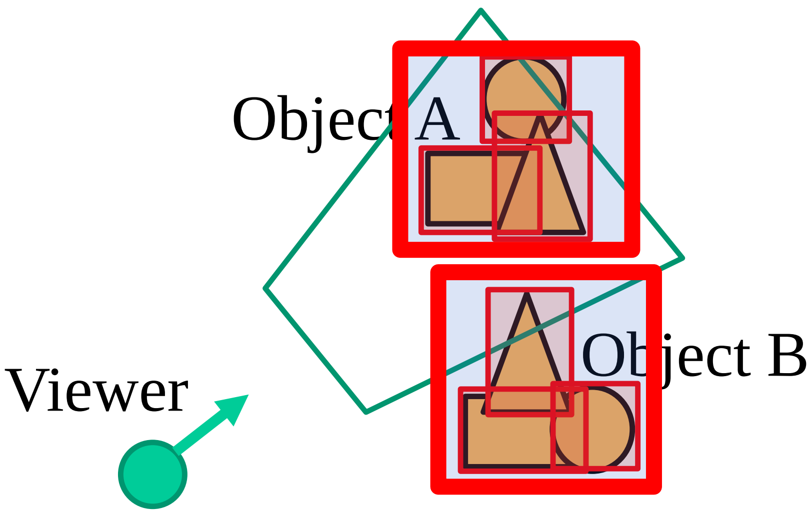
Frustum culling



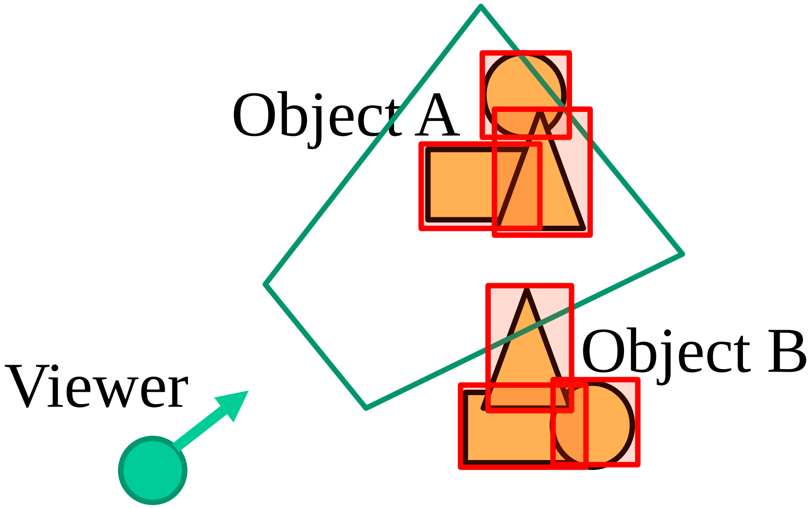
Frustum culling



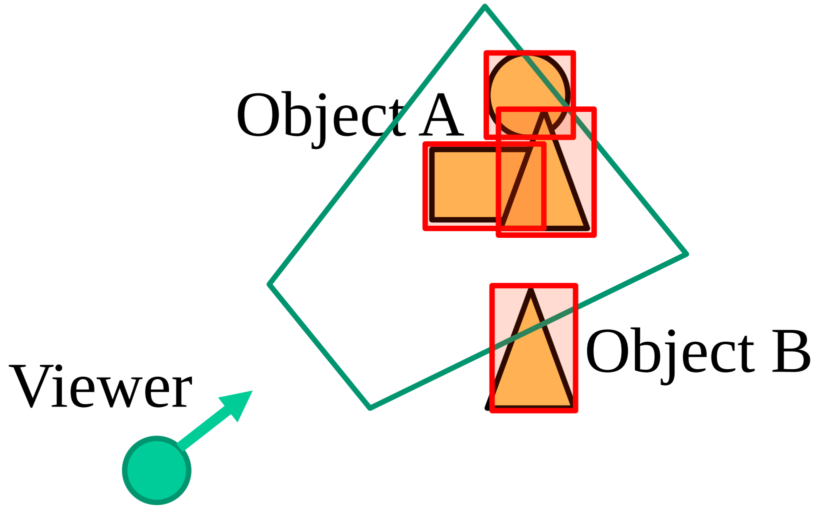
Frustum culling



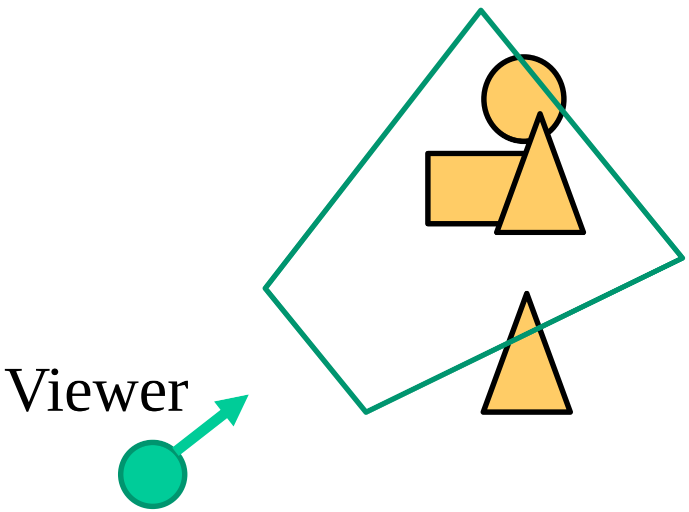
Frustum culling



Frustum culling



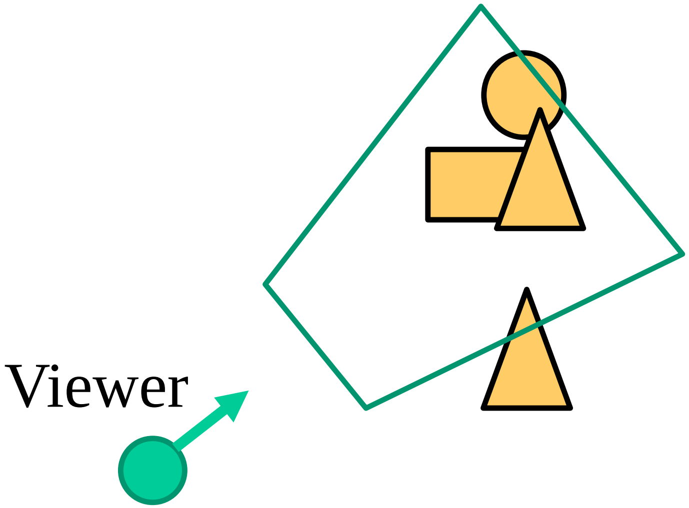
Frustum culling



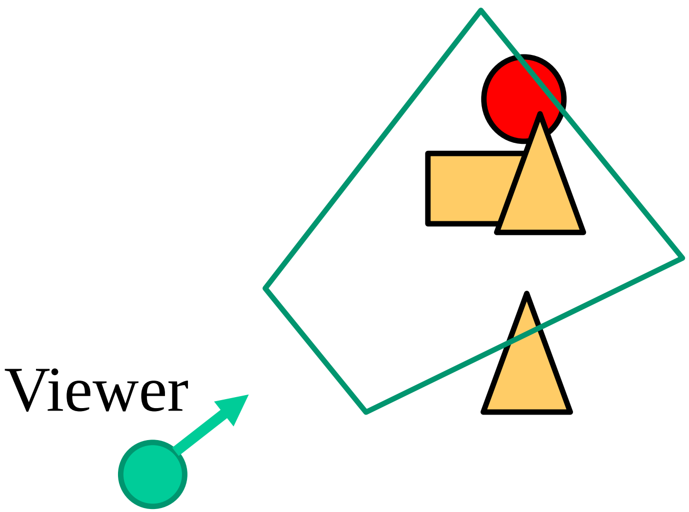
Culling

- Back-face culling
- Frustum culling
- Occlusion culling

Occlusion culling



Occlusion culling



Occlusion culling

Occlusion query: Rendering a bounding box for the object and counting how many pixels would pass the depth test.

However, no pixels are actually **written** during an occlusion query.

Occlusion culling

Occlusion queries are supported in (recent) hardware

```
glGenQueriesARB(1, &query);
glBeginQueryARB(GL_SAMPLES_PASSED_ARB, query);
      render_bounding_box(object);
glEndQueryARB(GL_SAMPLES_PASSED_ARB);

glGetQueryObjectuivARB(query,
         GL_QUERY_RESULT_ARB, &fragment_count);

if (fragment_count > 0) render_full(object);
```

Occlusion culling

The same hierarchical bounding volume structure is used for occlusion culling as for frustum culling.

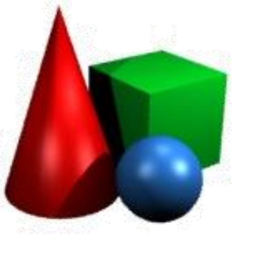
* Some tricks are needed for proper performance, though:
http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter06.html

Culling

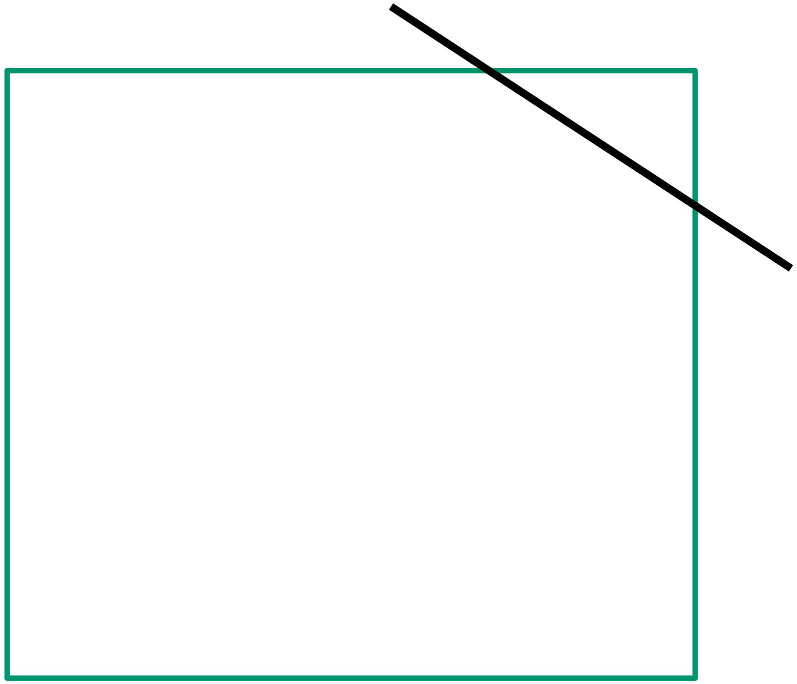
- Back-face culling
- Frustum culling
- Occlusion culling

Hidden Surface Removal

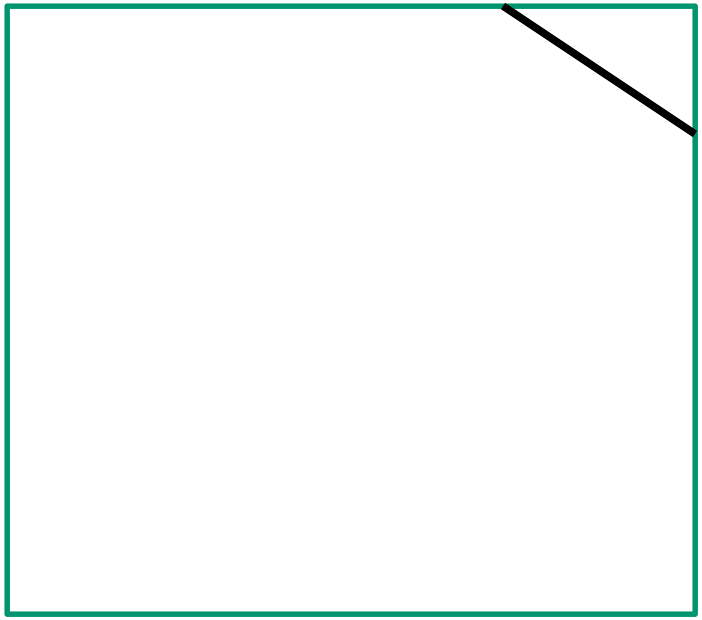
- **Sorting** Painter's algorithm | BSPs
- **Visibility tests** Z-buffer | Scanline algorithm
- **Culling** Back-face | Occlusion | Frustum
- **Clipping**



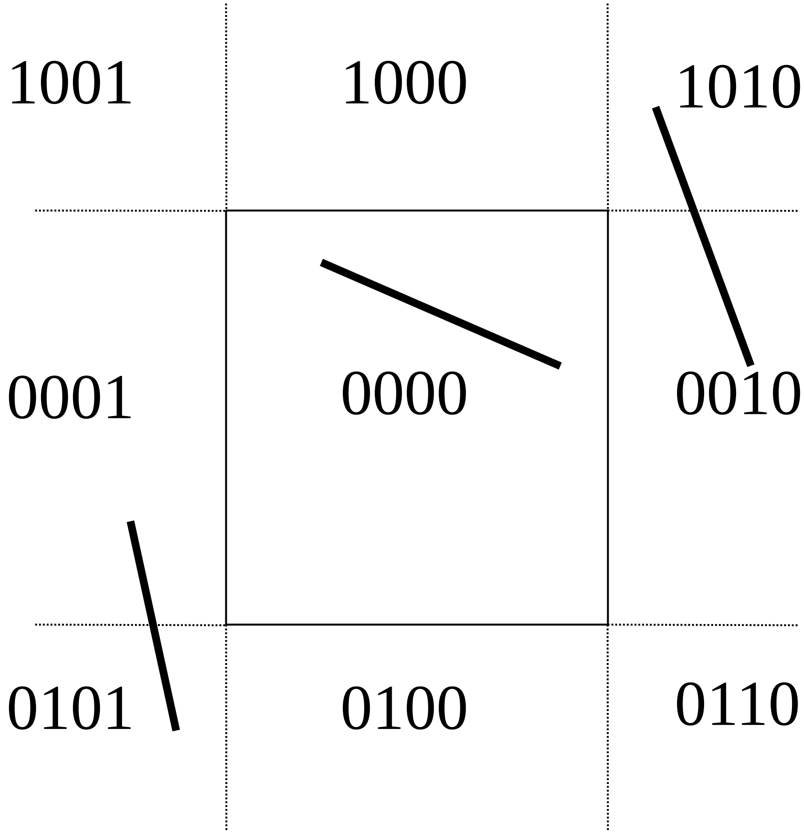
Line clipping



Line clipping

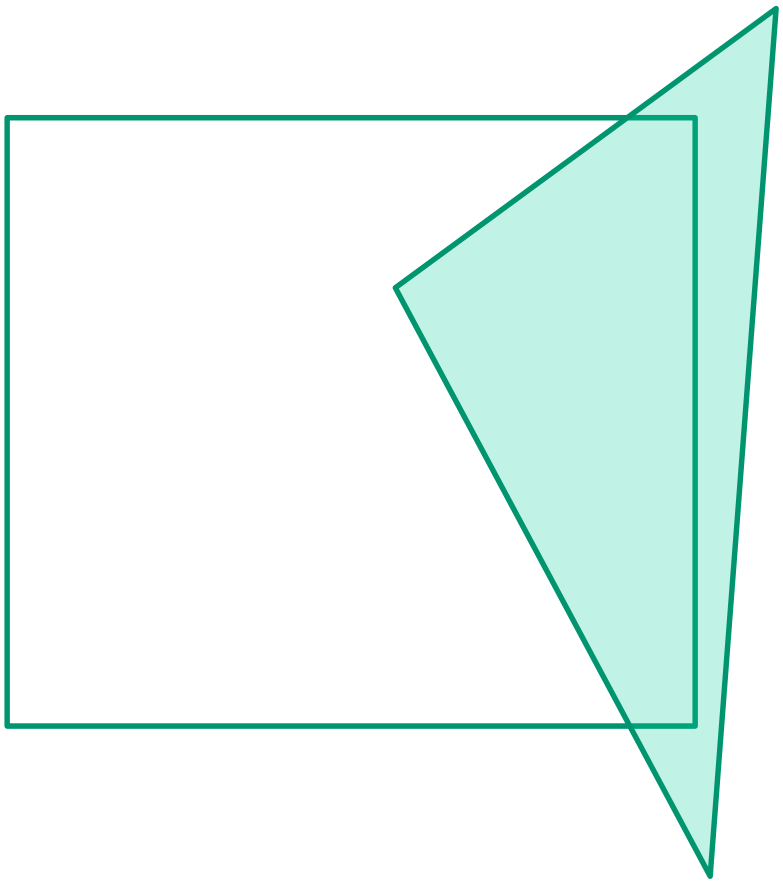


Cohen-Sutherland

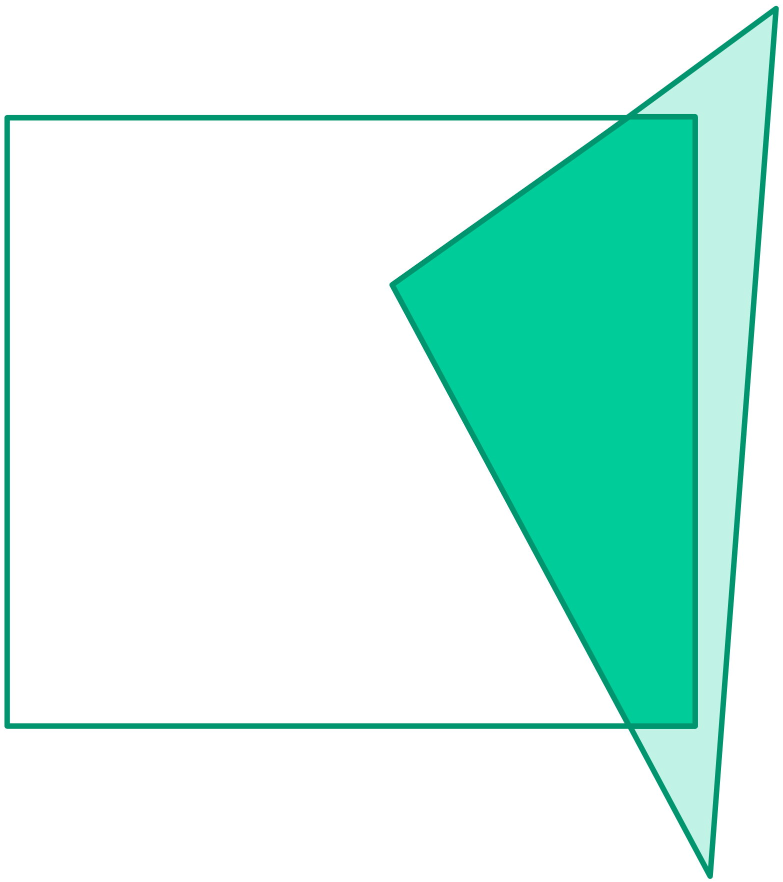


http://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm

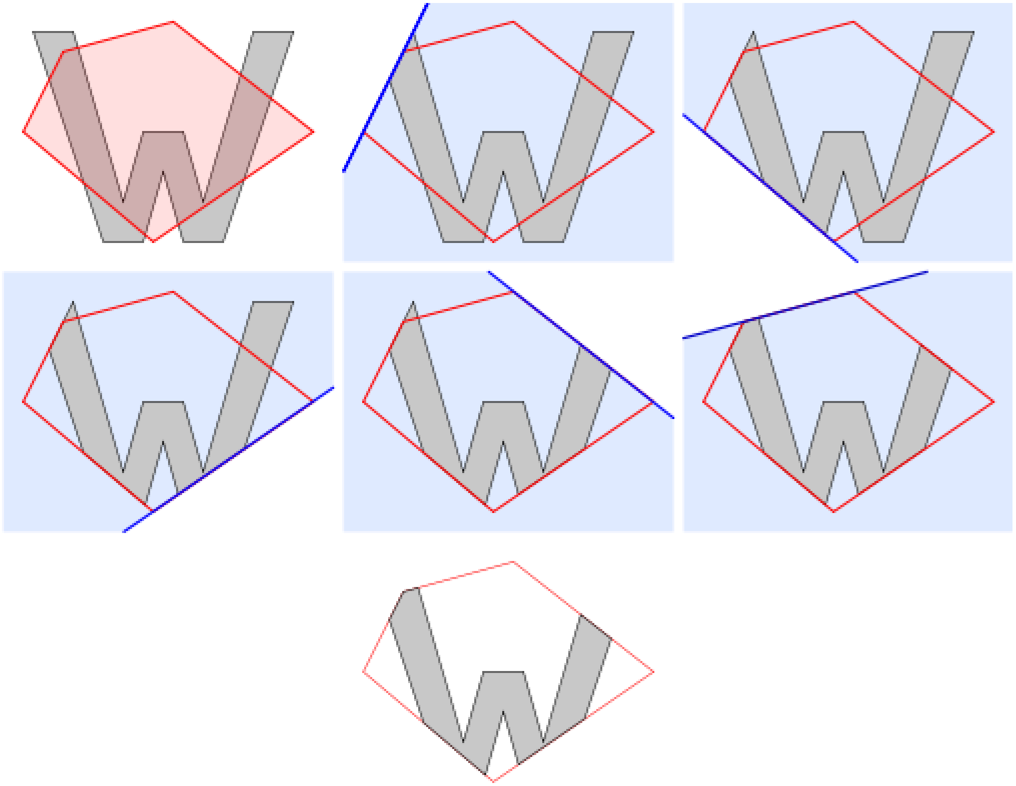
Polygon clipping



Polygon clipping



Sutherland-Hodgman



http://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm

Hidden Surface Removal

- **Sorting** — Painter's algorithm | BSPs
- **Visibility tests** — Z-buffer | Scanline algorithm
- **Culling** — Back-face | Occlusion | Frustum
- **Clipping** — Cohen-Sutherland | Sutherland-Hodgman

Frustum culling

Occlusion culling

Painter’s algorithm

Vertex transform

Culling and clipping

Normalized frustum culling

Back-face culling

Cohen-Sutherland

Sutherland-Hodgman

Rasterization

Fragment shading

Visibility tests & blending

Z-buffer

Standard Graphics Pipeline

Vertex transform

Culling and clipping

Rasterization

Fragment shading

Visibility tests & blending