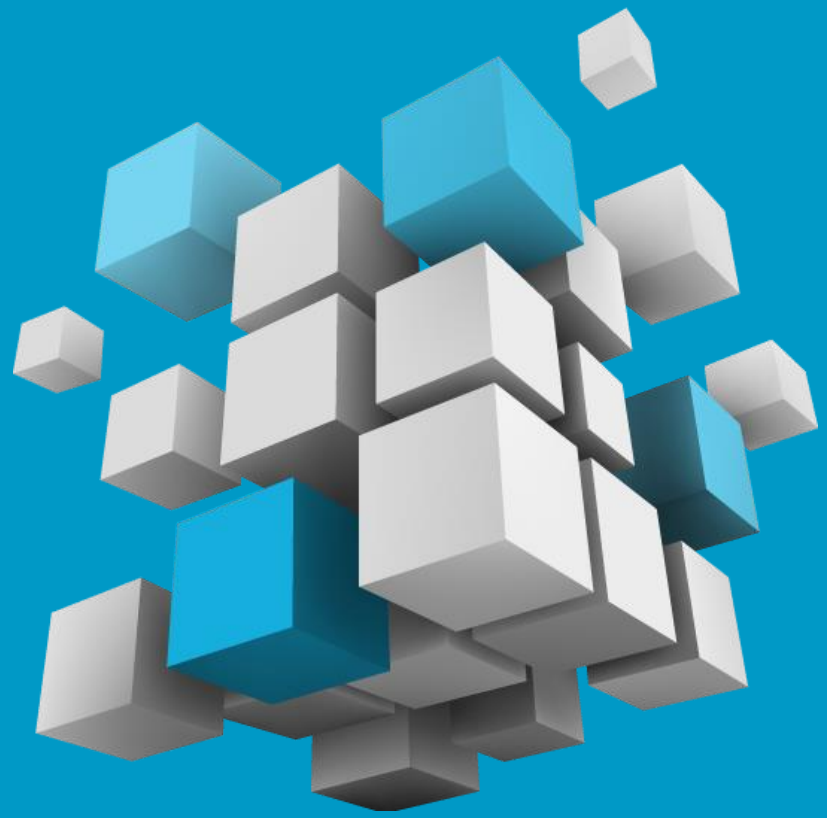
Real-time Graphics and Interaction

Modern graphics hardware



GPU vs. CPU

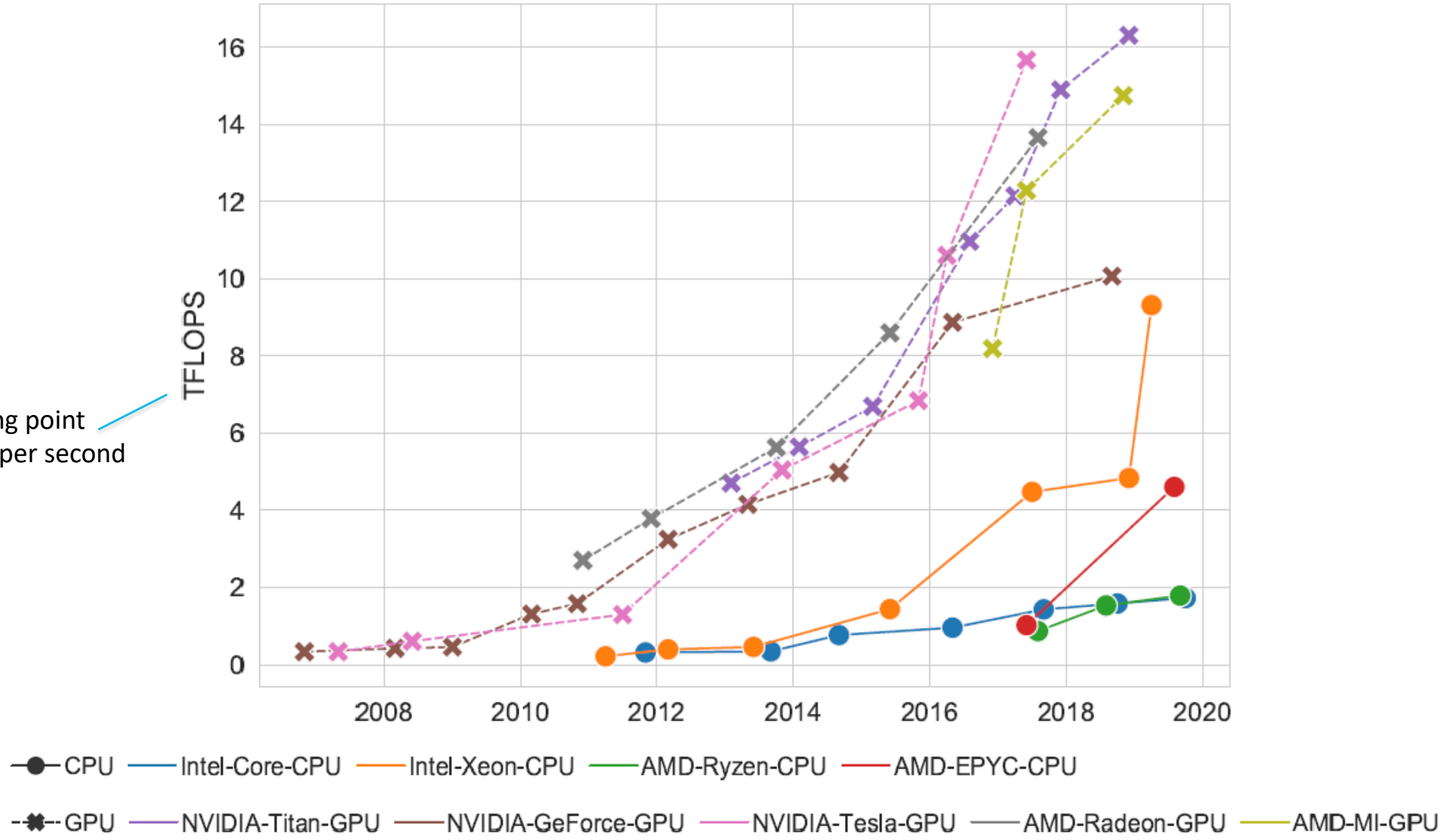
- CPUs typically come with small number of cores
  - Run code mostly in serial fashion
  - Limited SIMD processing (single instruction multiple data)
  - Local caching, branch prediction, …
  - General purpose

- GPU (graphics processing unit) have lots of shader cores (several thousands)
  - Massively parallel, SIMD processing
  - Optimized for throughput (latency hidden by switching to another fragment)
  - Focus on graphics but becoming more configurable

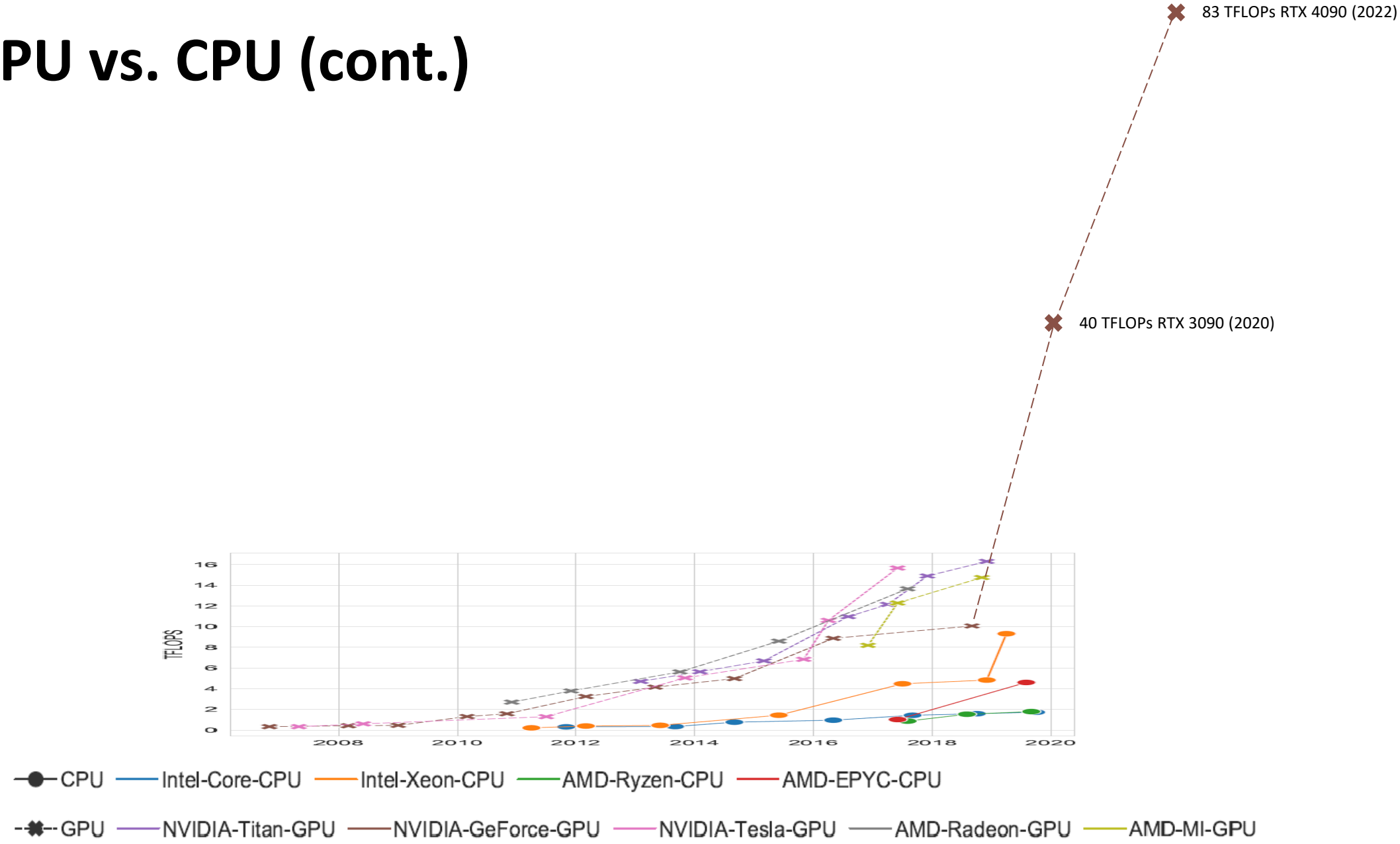
GPU vs. CPU

[Sun 2020]

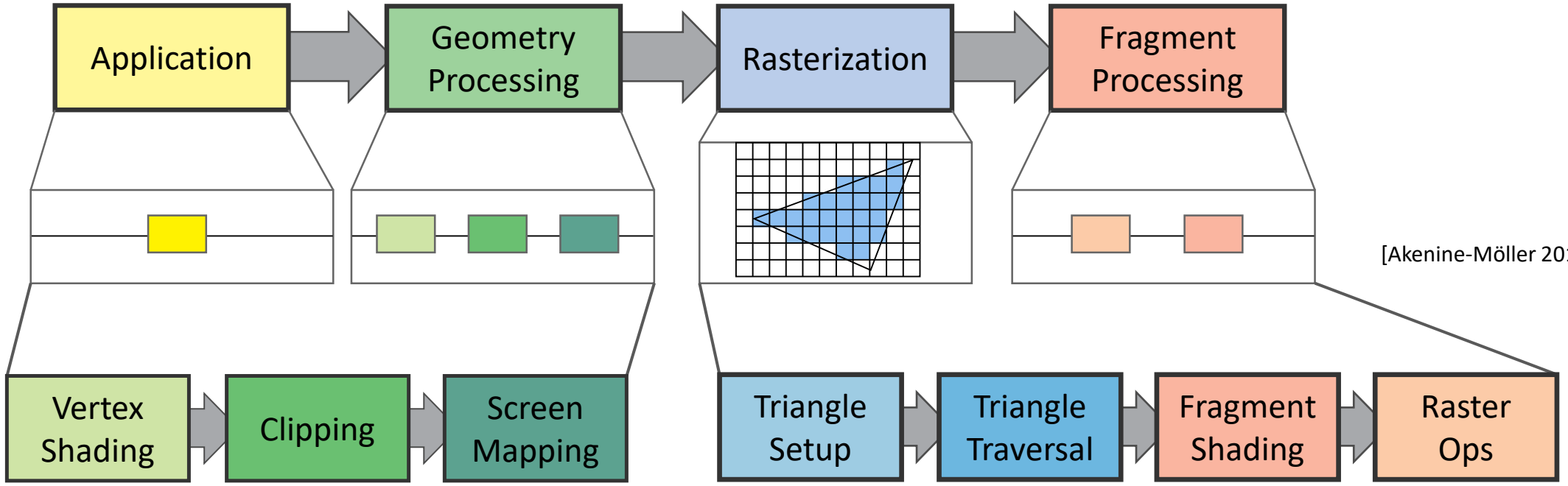
10^{12} floating point operations per second



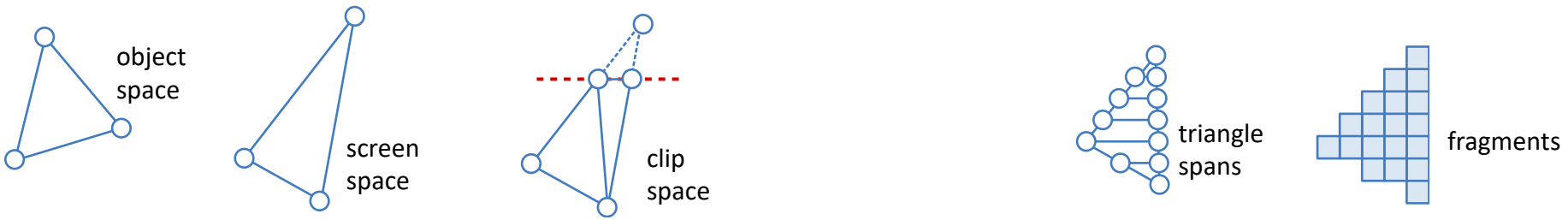
GPU vs. CPU (cont.)



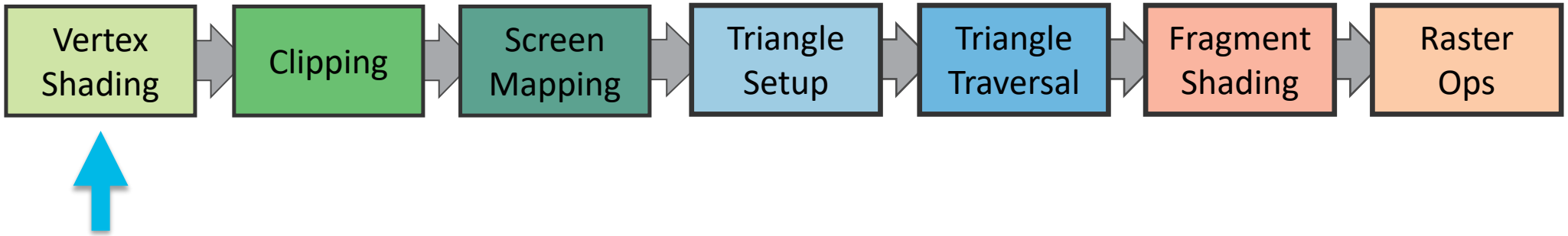
Graphics Rendering Pipeline

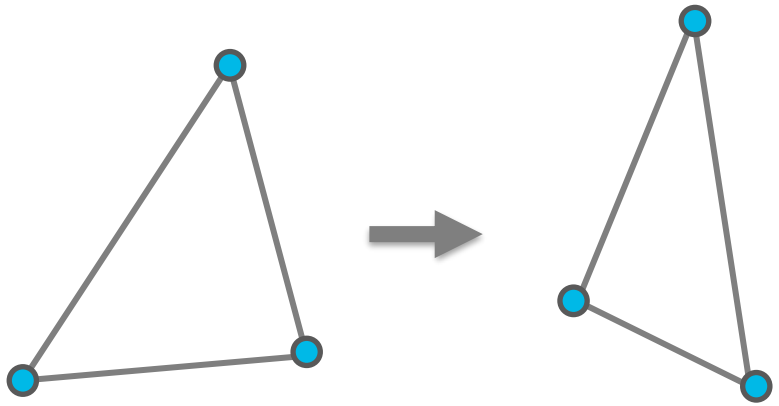


[Akenine-Möller 2018]



Rendering a Triangle

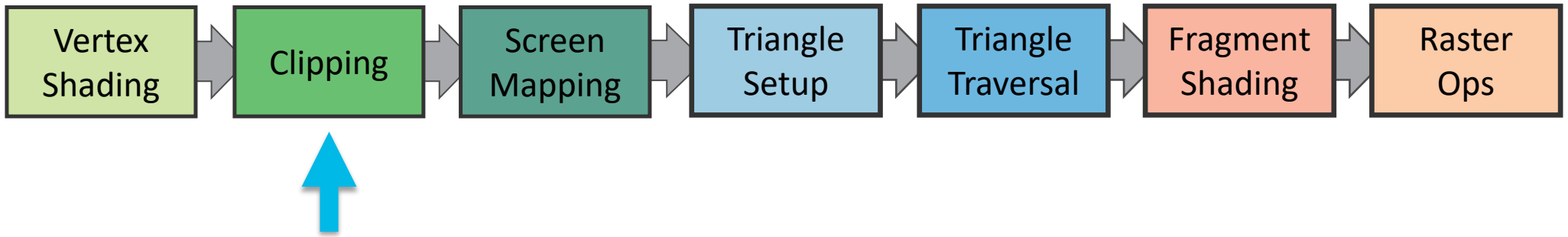


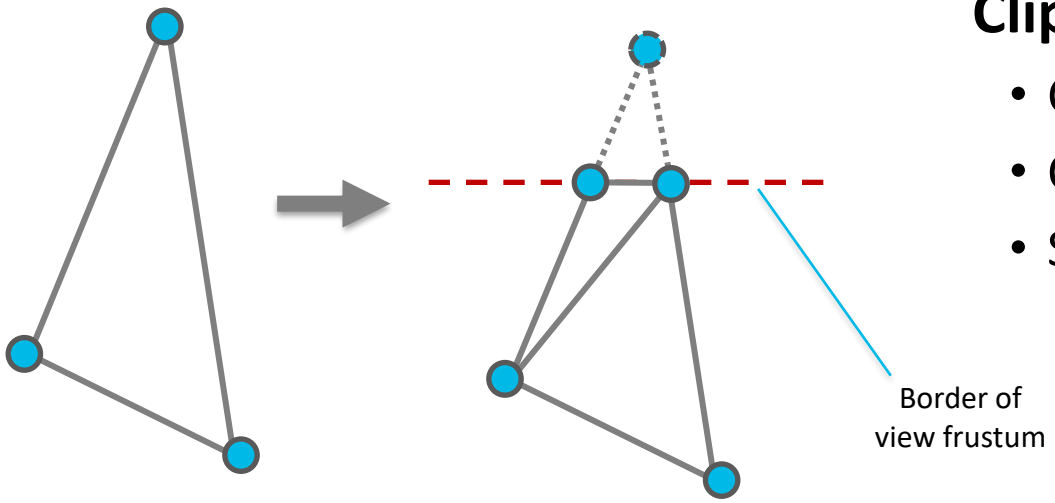


Vertex shading

- Transform vertices
- Assign vertex attributes like position, color, normals, …

Rendering a Triangle (cont.)

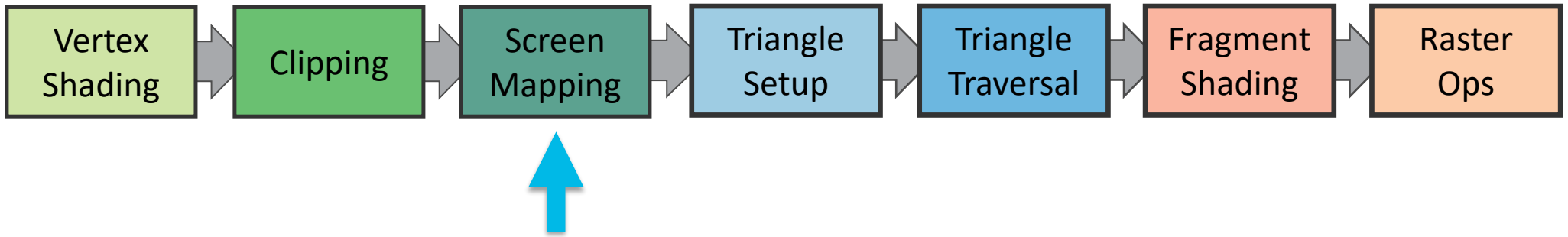




Clipping

- Clip triangle against view frustum
- Cut off anything outside screen space
- Split into new triangles (if necessary)

Rendering a Triangle (cont.)

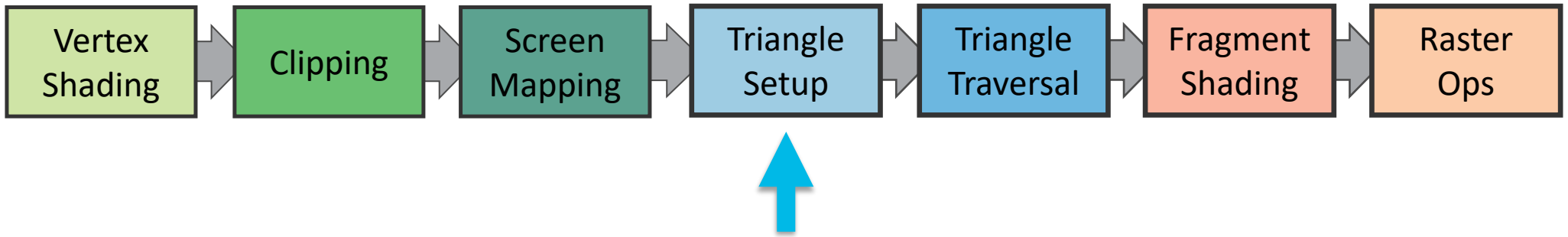


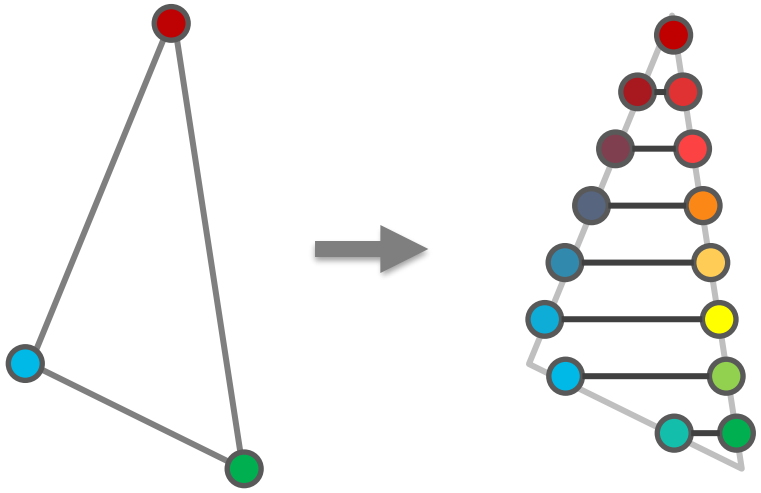
Screen mapping

- Perspective projection into screen coordinates
  - Division by w (homogenization)

4^{th} vector component

Rendering a Triangle (cont.)

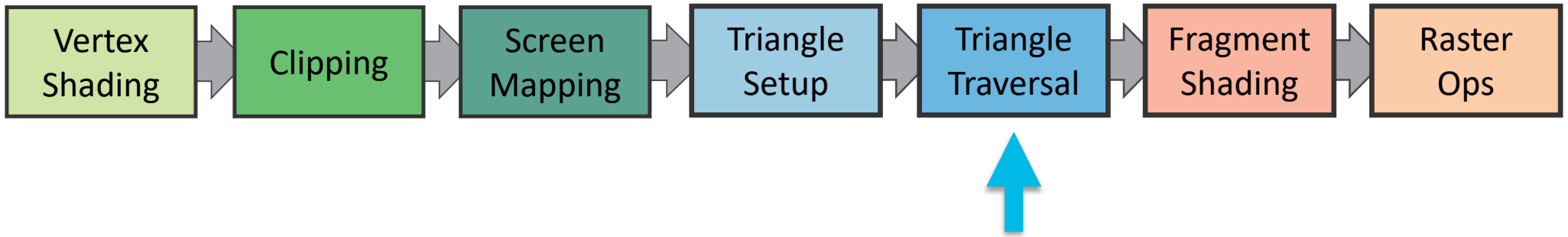


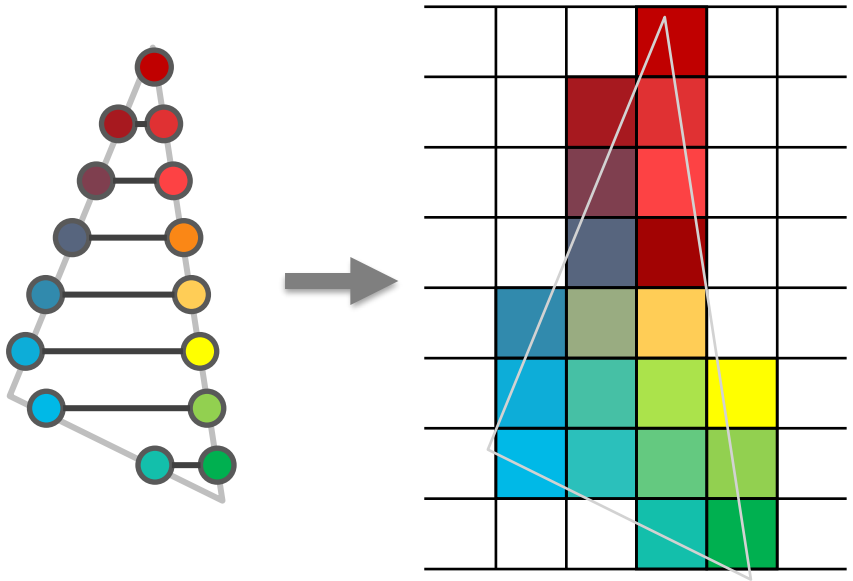


Triangle setup

- Interpolate vertex attributes along triangle edges
- Split triangles into one-pixel wide lines

Rendering a Triangle (cont.)

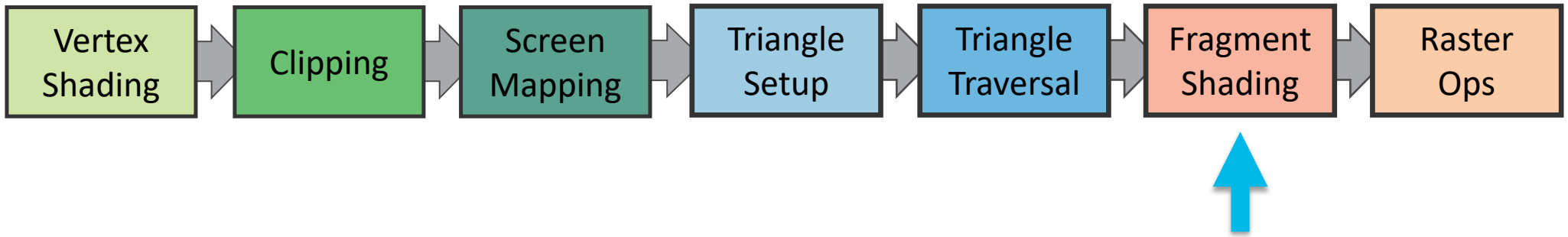


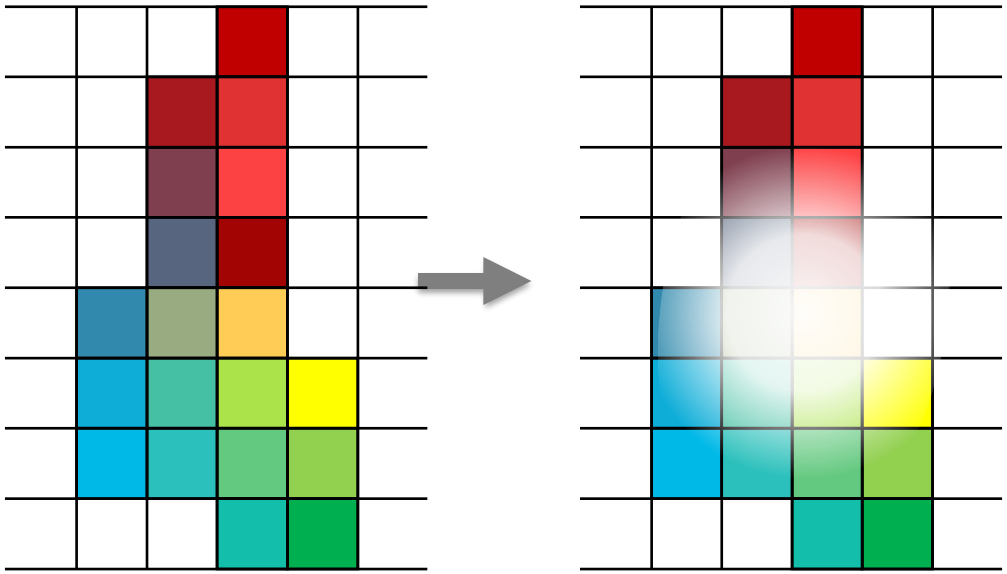


Triangle traversal

- Interpolate attributes along lines
- Create fragments

Rendering a Triangle (cont.)

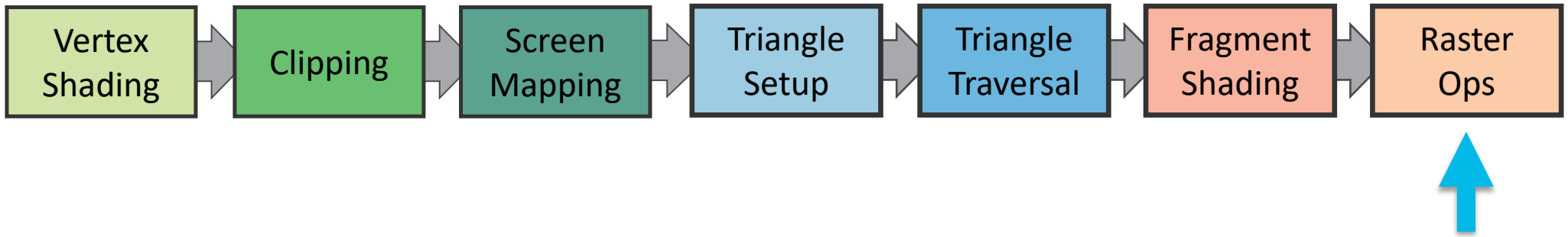


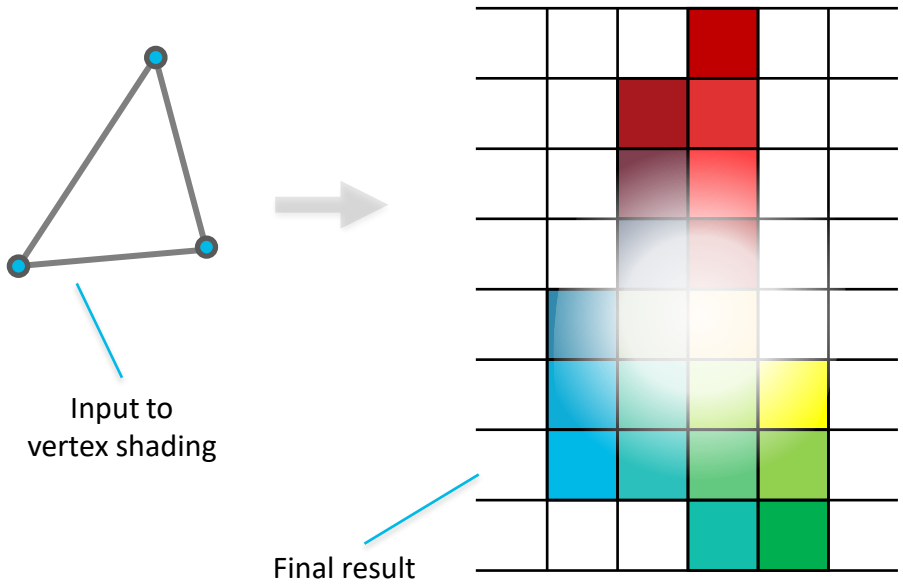


Fragment shading

- Process each fragment
- Determine final color

Rendering a Triangle (cont.)

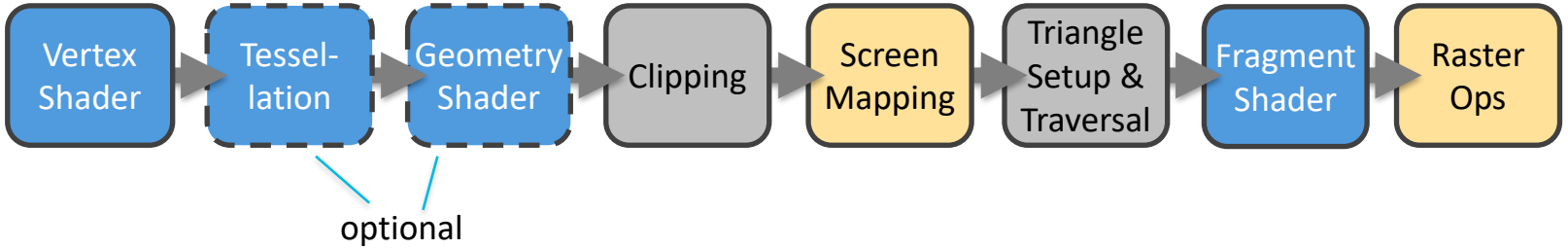




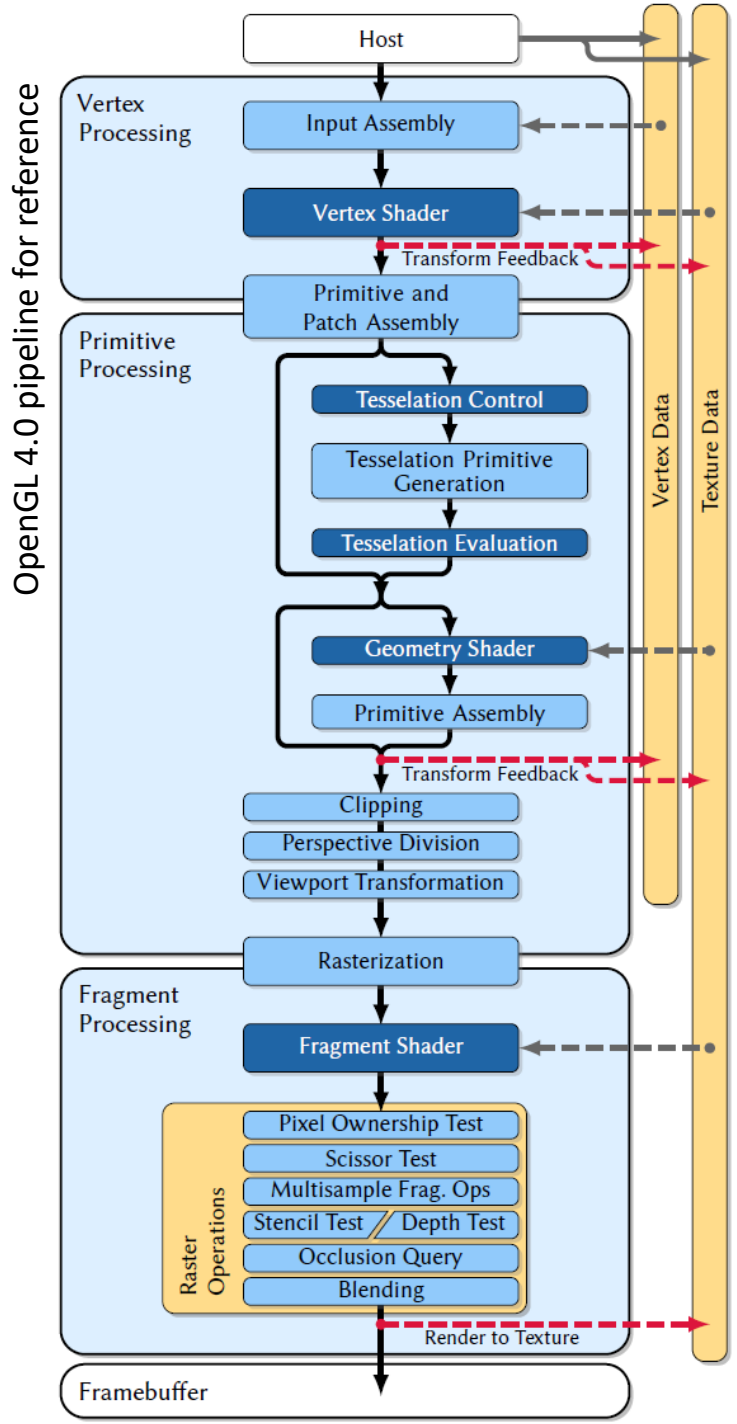
Raster operations

- Determines whether fragment is rendered into screen pixel
- Test each triangle fragment
  - Depth test (fragment occluded?)
  - Alpha test and blending
  - Scissor test
  - …

OpenGL Pipeline

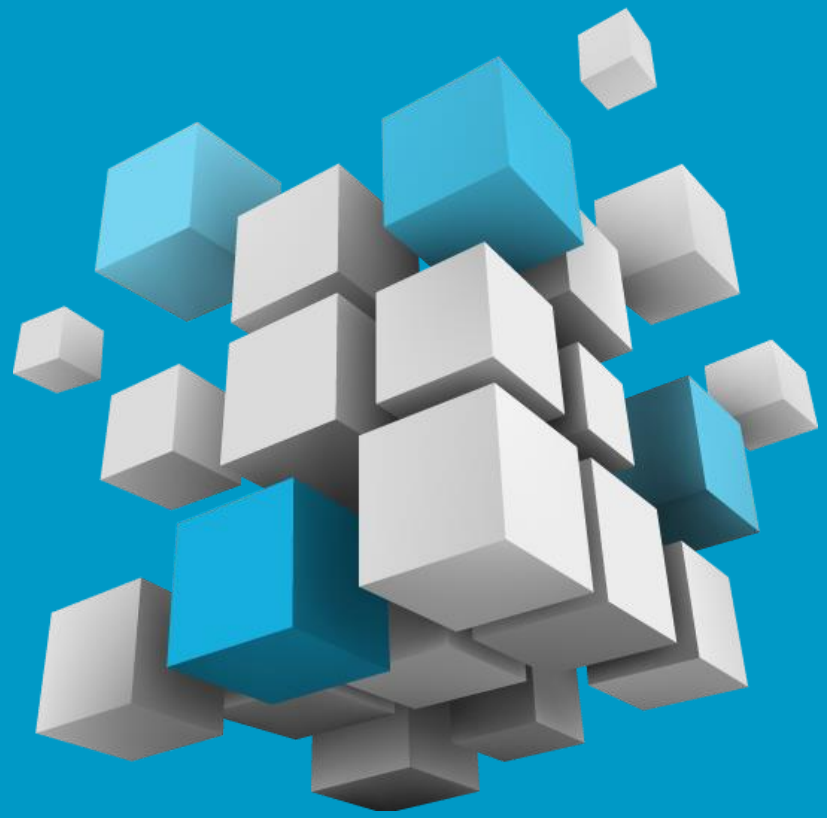


- Some parts are fully programmable (shaders)
- Some configurable
  - Screen mapping (viewport)
  - Blending, depth test, scissor test, … (Raster Ops)
- Some fixed
  - Clipping & rasterization



Real-time Graphics and Interaction

Rendering & shading



Shaders

Small programs

Run per geometry element or per fragment

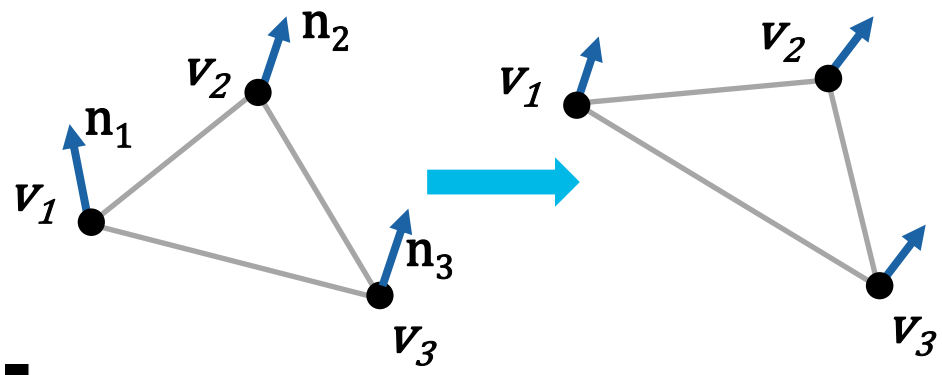
Can access texture memory

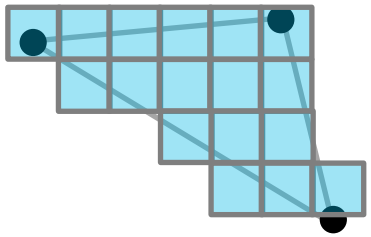
Pixel
small colored square on the screen

Fragment
rasterizer creates one fragment per pixel; holds additional information like interpolated vertex attributes

```
for each vertex in geometry {
    transform vertex
    assign color
    ...
}
```
} **Vertex shader**

```
for each fragment in triangle {
    calculate lighting
    assign color
    ...
}
```
} **Fragment shader**





Vertex Shader

Processes individual vertices

Input

- Vertex attributes like position, color, normals, …
- Bound via a vertex array object (VAO)
  - Holds vertex buffer objects (VBOs) with the actual data
- Specified with

```
layout(location = 0) in vec4 position;
layout(location = 1) in vec4 color;
```

Output

- Default: `out vec4 gl_Position;`
- Varying output variables defined as

```
out <type> <name>;
```

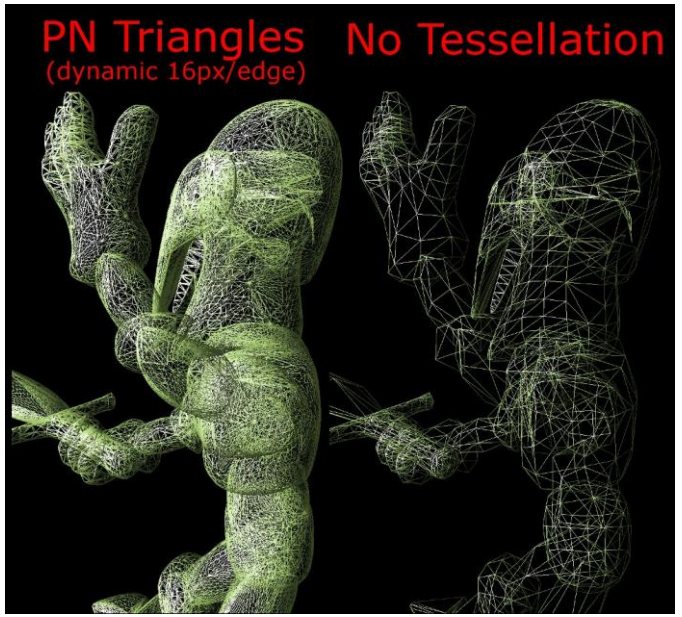
```
#version 330 core
in vec4 in_Vertex;
in vec3 in_Normal;
in vec4 in_Color;
in vec3 in_TexCoord;

uniform mat4 dataToClip = mat4(1);

out vec4 color_;
out vec3 texCoord_;

void main() {
    color_ = in_Color;
    texCoord_ = in_TexCoord;
    gl_Position = dataToClip * in_Vertex;
}
```

Tessellation Shader (optional)



Subdivides input patches adaptively into smaller primitives

Uses three stages

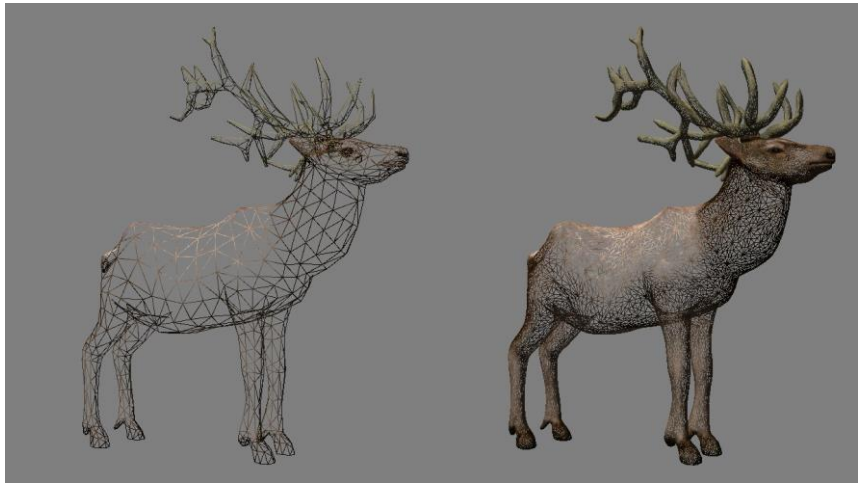
- Tessellation control
- Primitive generation (fixed)
- Tessellation evaluation

Input

- Patches from vertex shader (array of vertices)

Output

- Generates new primitives:
  points, lines, or triangles



Geometry Shader (optional)

Processes output of vertex/tessellation shader and can create more primitives

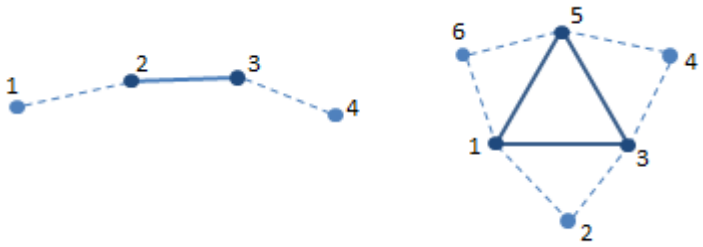
- Has full knowledge about primitive and adjacency

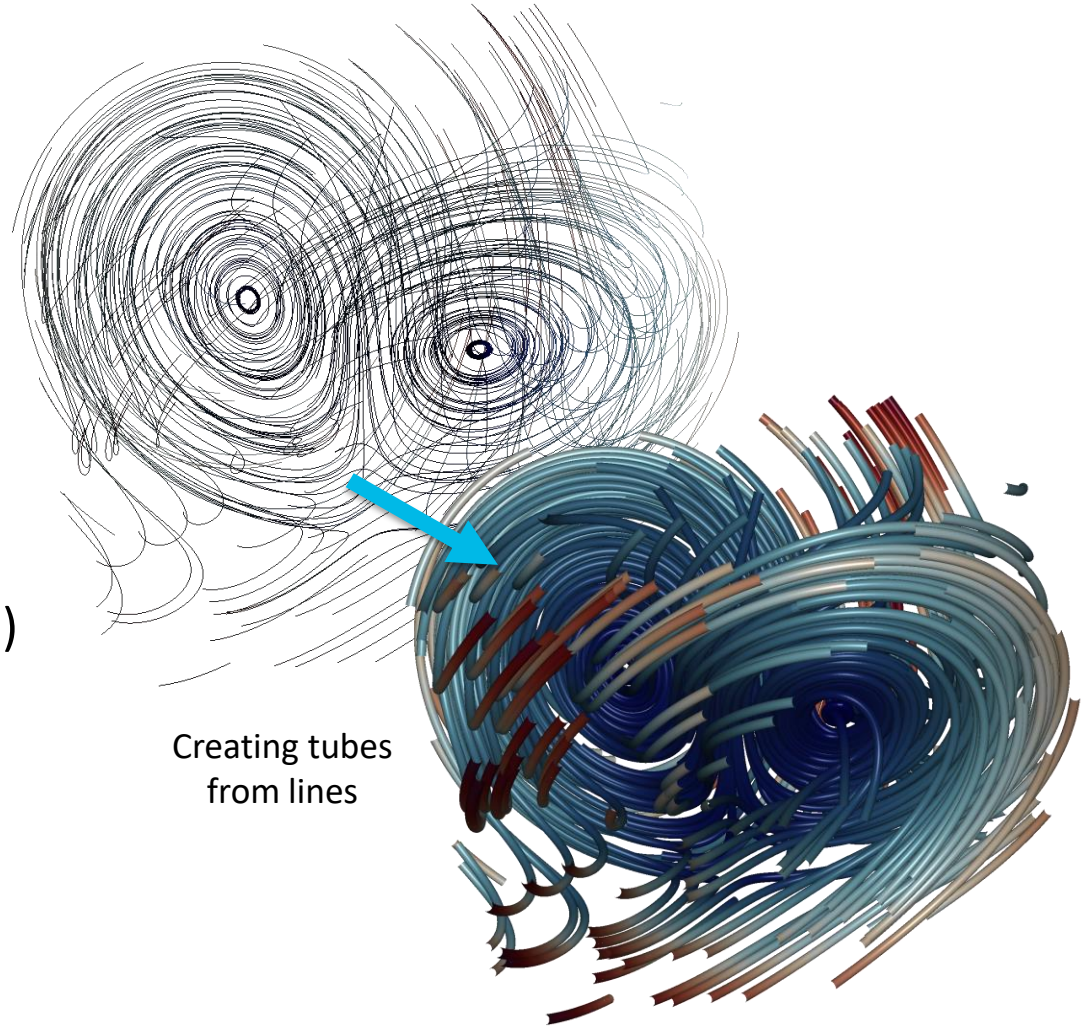
Input

- Assembled primitives (lines, triangles, triangle strips, …)

Output

- Zero or more primitives





Creating tubes from lines

Fragment Shader

```
1  #version 150
2
3  out vec4 colorOut;
4
5  void main() {
6      colorOut = vec4(1.0, 0.0, 0.0, 1.0);
7  }
8
```

Processes fragments

- Can only access current fragment
- No framebuffer access

Input

- Single fragment with window position (x,y) and depth
  `gl_FragCoord`
- All varying attributes from previous stage

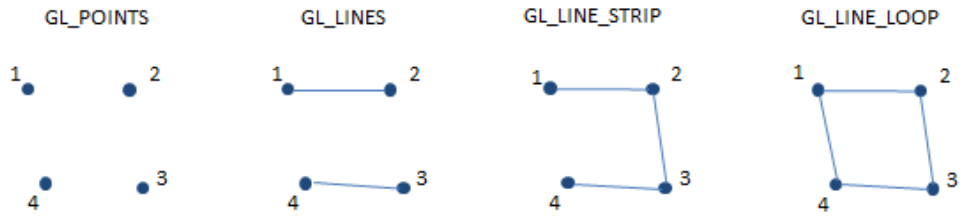
Output

- Renders data into default framebuffer
  usually, color and depth
- Can render into up to 8 textures when
  using framebuffer objects

Scalars or vectors

OpenGL Primitives

(see also lecture on Modeling – polygon meshes)

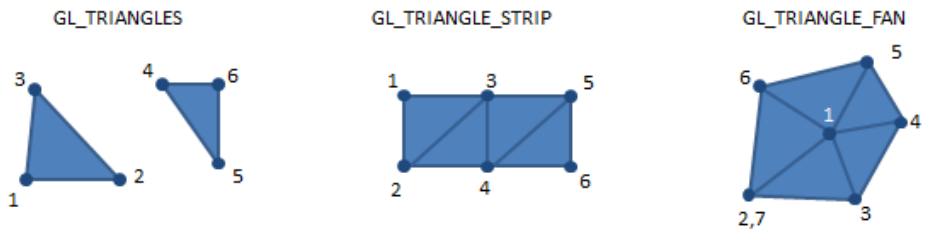


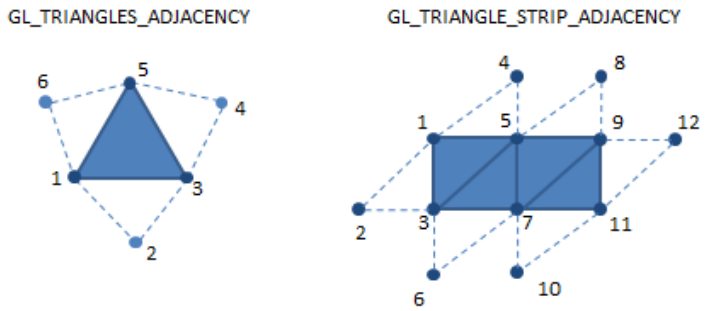
Vertex attributes describe only points

Connectivity information needed for rendering

- Provided when calling `glDrawArrays`, `glDrawElements`, …

Adjacency used by geometry shader





Shading

Flat shading

- `flat out vec4 color;` and `flat in vec4 color;` in shaders

Gouraud shading

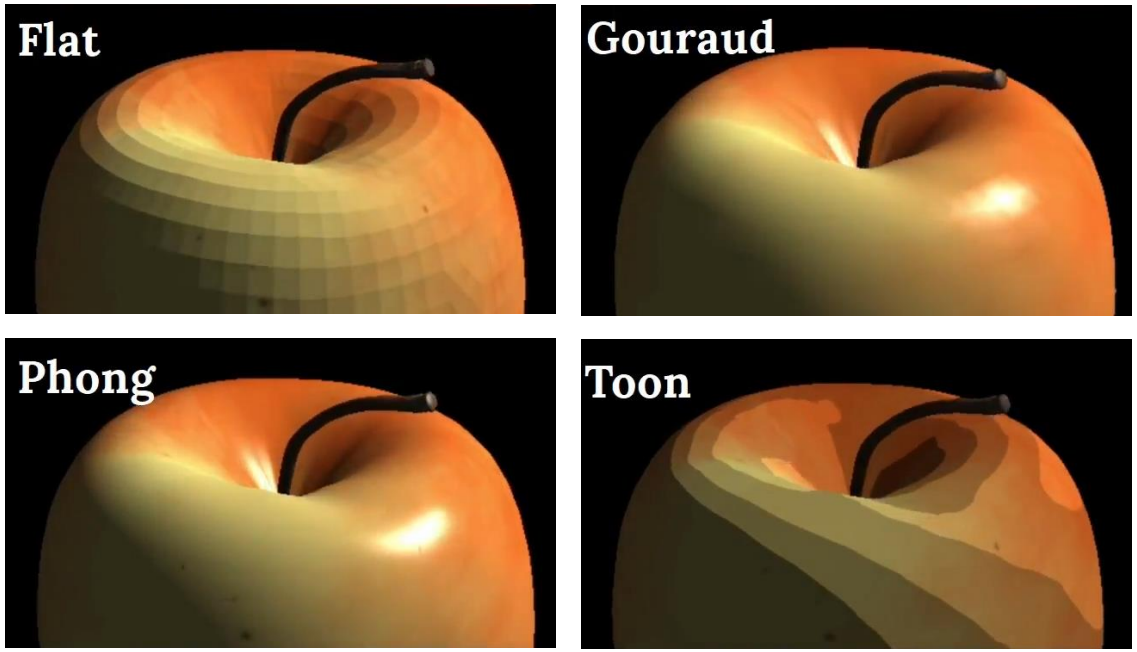
- Computation in vertex shader
- Result interpolated for each fragment

Phong shading

- Interpolate vertex attributes (position, normal, color, …)
- Computation in fragment shader

Cel shading / Toon shading

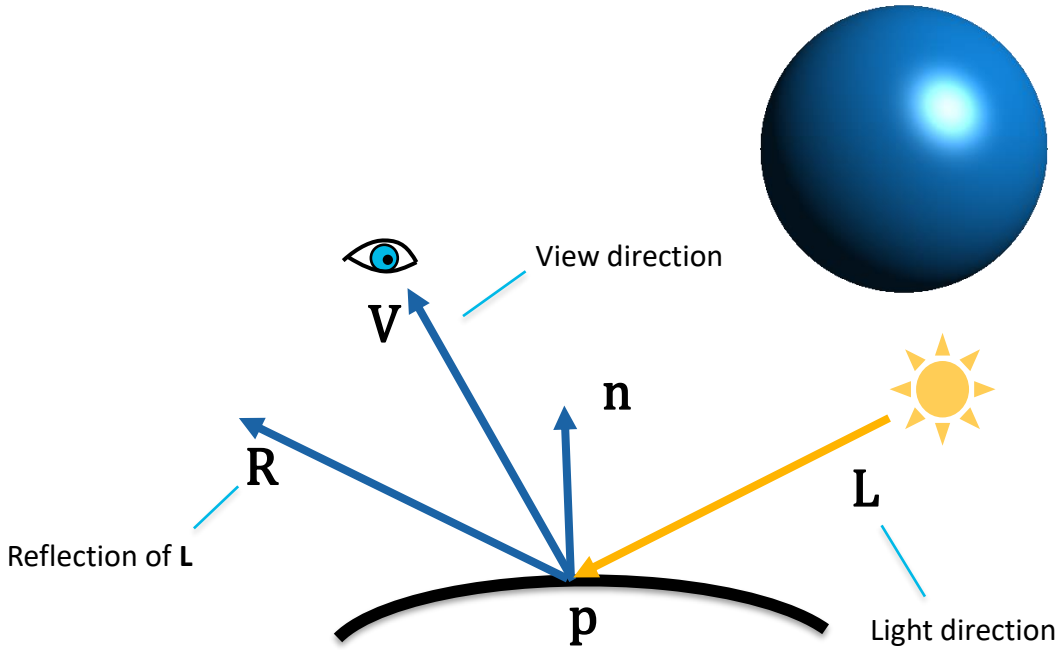
- Compute smooth lighting
- Quantize into discrete shades



[YouTube]

Phong Illumation

(Lectures on Illumination / Materials)



Shading only incorporates local information

- Surface normal, directions to camera and light source, depth, …
- Fast!

Phong illumination model [Phong 1975]

- If **V** close to **R**, the specular highlight gets stronger

$$I_p = k_a + \sum_{m \in lights} k_d(\mathbf{L_m} \cdot \mathbf{n}) + k_s(\mathbf{R_m} \cdot \mathbf{V})^{\alpha}$$

Illumination for point **p**

Ambient term

Diffuse term

Specular term (highlight)

Illumination

(Lectures on Illumination / Materials)

Note
vectors (directions) need to be **normalized** for correct results!

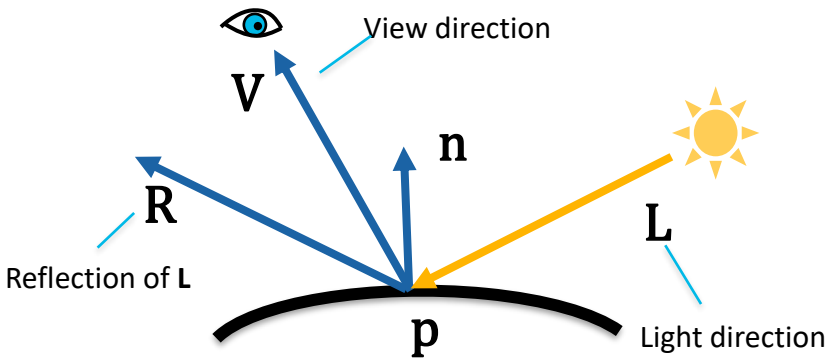
Required input depends on material and illumination model

- Color, normal, position, light parameters, view parameters, …

Decide on which coordinate system to use

- Global coordinates vs. local coordinates

Pick one and stick to it



Local coordinates:

- Forward initial vertex position (untransformed)
- Transform light and camera position using $\mathbf{M}^{-1}$
- Compute $\mathbf{L}$ and $\mathbf{V}$ using $\mathbf{p}$ and light/camera positions

Global coordinates:

- Transform normal with **normal matrix**
- Compute $\mathbf{L}$ from transformed vertex and light position

Transparency

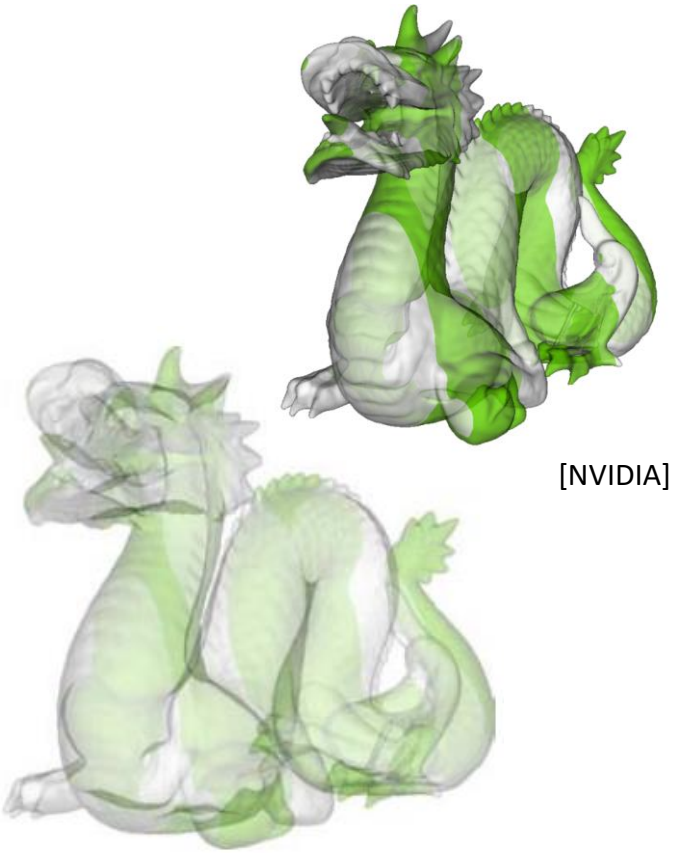
Encoded as alpha value (opacity) in RGBA colors

OpenGL performs compositing after fragment shader

- Blending function determines final compositing (alpha blending)
- Back to front blending:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Correct blending requires depth sorting



[NVIDIA]

Alternative:

Order-independent transparency (Depth Peeling)

- Render scene multiple times, capture one layer at a time
- Blend front to back

Screen Space Ambient Occlusion (SSAO)

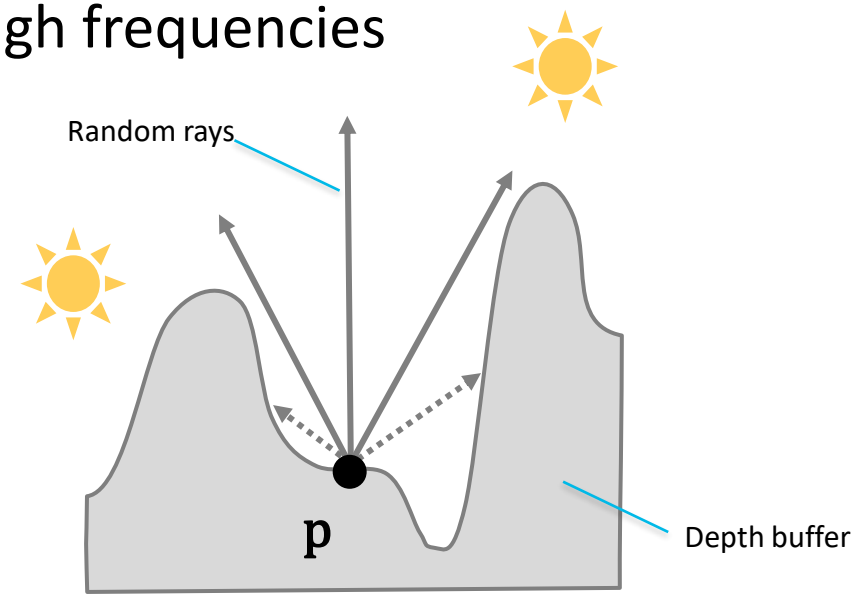
Coarse approximation of global illumination

- Screen-space approach
- Requires only depth buffer

Compute average visibility per pixel (occlusion factor)

- Create random sample rays in local neighborhood (~16)
- Check if ray is occluded by depth map

Post processing blur removes high frequencies



And more…

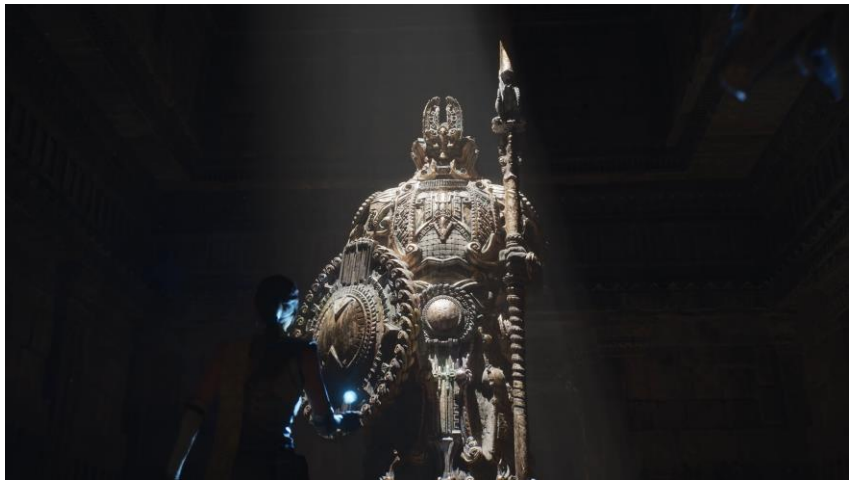
Shadow mapping (Lecture on Illumination)

- **Idea:** things not visible from a light are shadowed

[GPU Gems 2]

High dynamic range (HDR) & Tone mapping

- Use float precision to cover higher contrast ratios like human vision
- Tone mapping compresses HDR back to low dynamic range

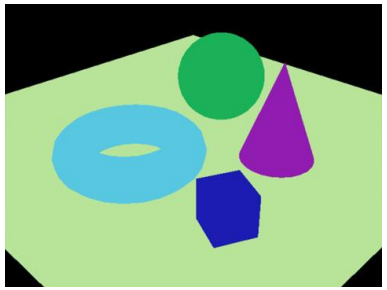


[Unreal Engine 5, YouTube]

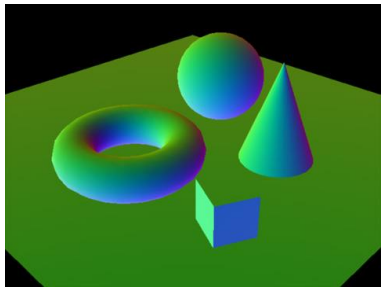
And more…

Deferred Rendering / Deferred Shading

- Decouple lighting from scene geometry
  (lots of geometry + expensive shading)
- Render first, compute shading & illumination in second pass

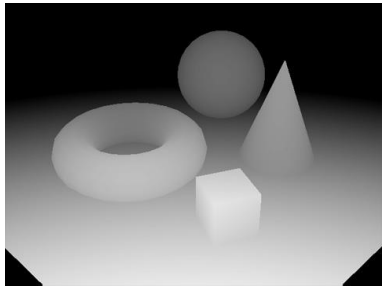


Diffuse color G-Buffer

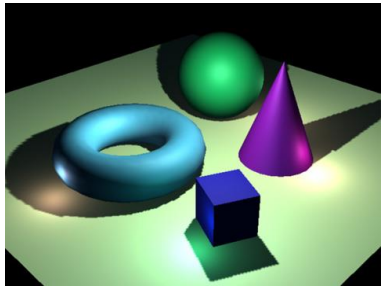


Normals G-Buffer

[wikipedia.org]



Depth

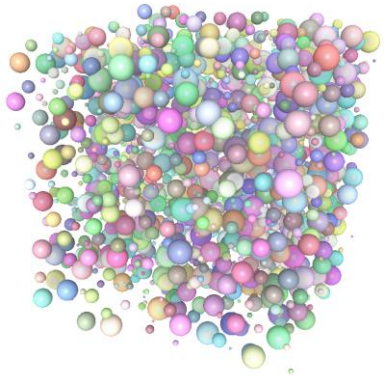


Result after 2nd pass

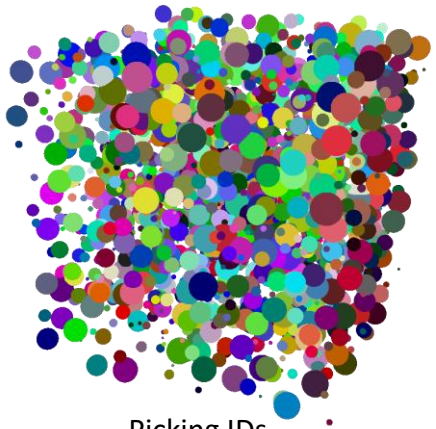
Picking – selecting objects on screen

- Render object IDs into separate buffer
- Read back values from GPU

…



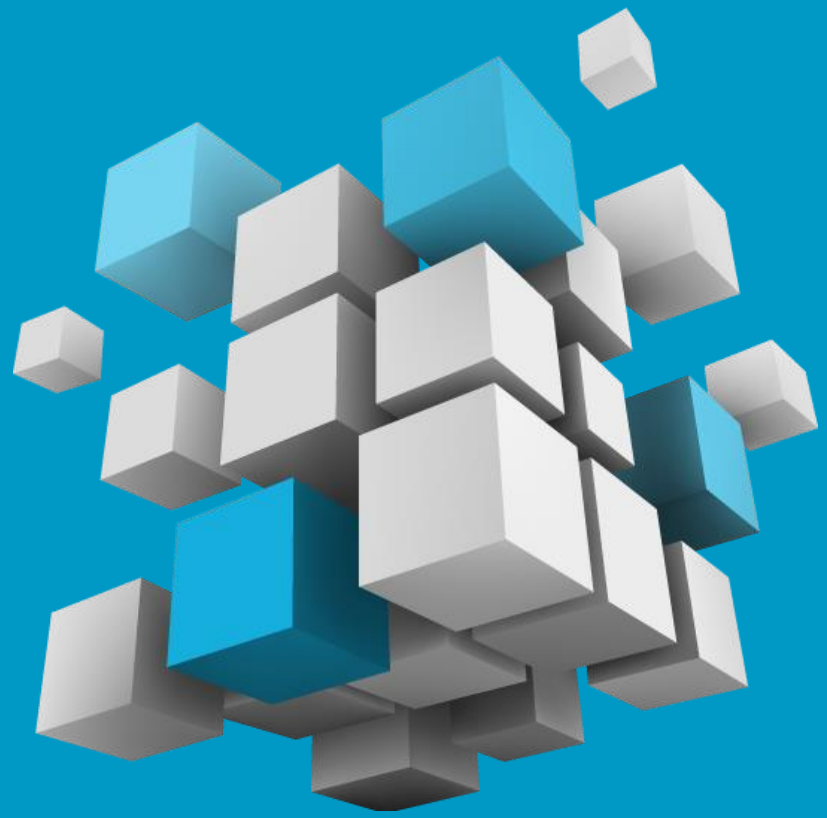
Rendered scene



Picking IDs

Real-time Graphics and Interaction

Data structures for large scenes



Large Scenes

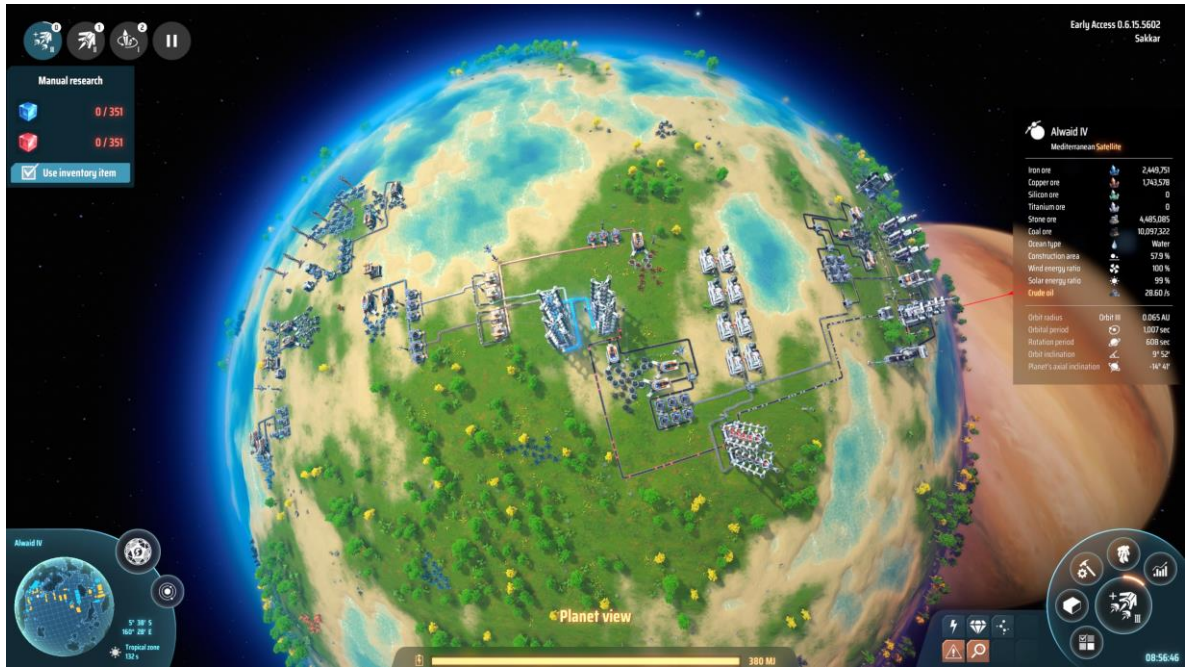
Scenes can contain hundreds of thousands of objects

- Several million triangles not uncommon

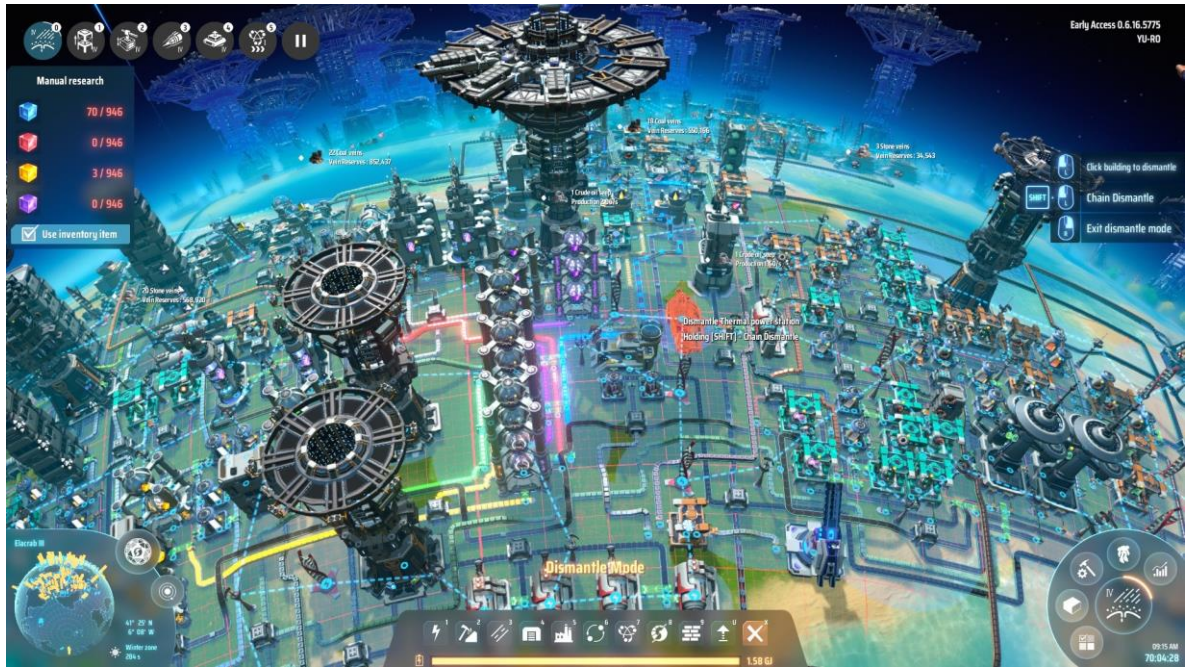
Need to

- quickly locate objects at a position
- identify visible objects
- detect collisions
- sort objects according to distance (draw order)
- …

Also relevant in offline rendering



[Dyson Sphere Program]



Level of Detail (LoD)

Show & render details only when needed

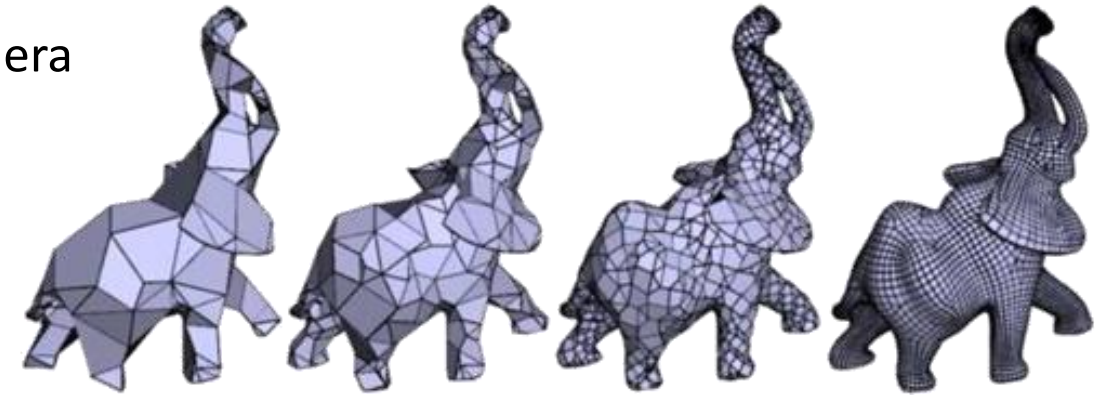
Provide different resolutions (levels) for

- Meshes
- Textures
- Light maps
- …

Use mipmaps for automatic LoD in GLSL shaders (see Lecture on Materials)

Select appropriate level based on

- Distance to camera
- Size on screen
- Importance



[Maglo et al. 2012]

[atangeo.com]

Spatial Data Structures

Organize objects in space for efficient lookup

- Also known as *acceleration structures*
- Often relying on bounding boxes

Object partitioning (BVH)

- Divide objects into disjoint groups
- Groups might overlap

Space partitioning (BSP, Octree)

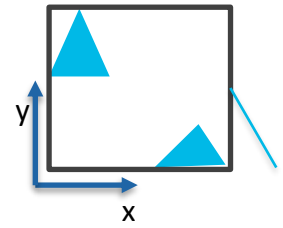
- Regular (uniform subdivision)
- Irregular
- Objects might be part of more than one partition

[pcgamesn.com]

Bounding Volume Hierarchy (BVH)

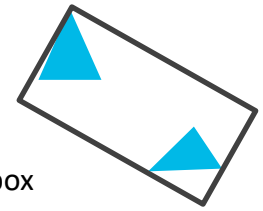
Bounding volume encloses all objects it contains

- Bounding box or bounding sphere

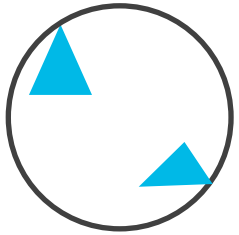


Bounding box

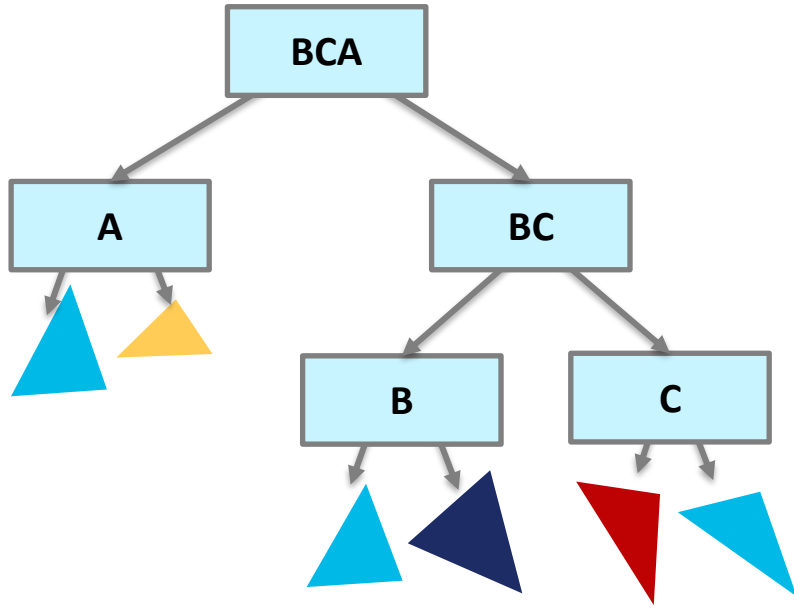
Axis-aligned bounding box (AABB)



Object-aligned bounding box (OOBB)



Bounding sphere



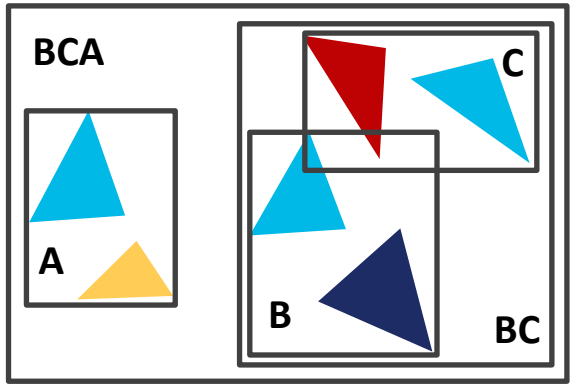
BVH of the scene

Leaves of BVH tree represent objects

Group close objects together (often 2)

- Allowed to overlap

Traversal needs to check bounding boxes of both subtrees



Scene

BSP Trees (Binary Space Partitioning)

Nodes in the tree represent cutting planes

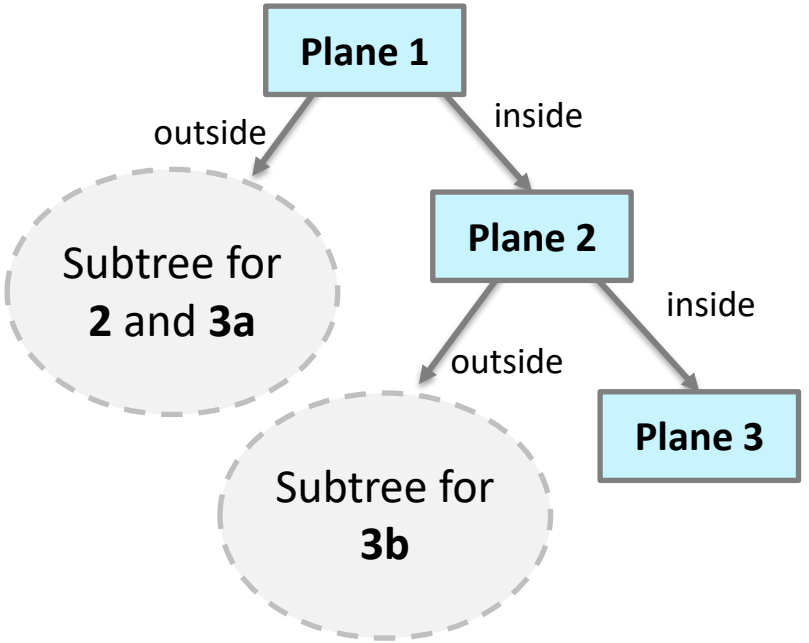
- Oriented half spaces (inside & outside)
- Left and right subtree

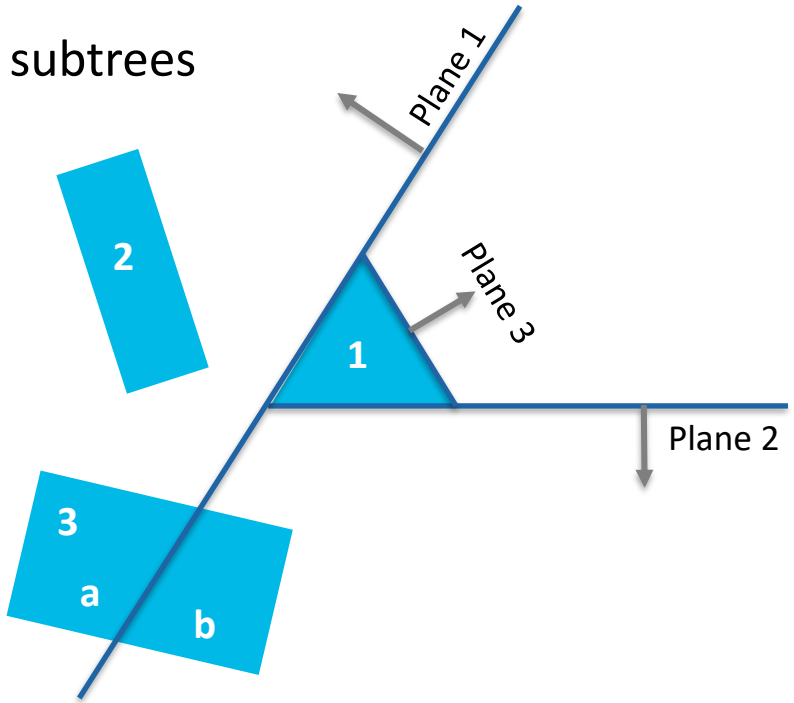
Each subtree contains **all** objects on its side of the cutting plane

- Cut objects are part of both subtrees

BSP trees are either

- Axis-aligned or
- Polygon-aligned





Axis-aligned BSP Trees

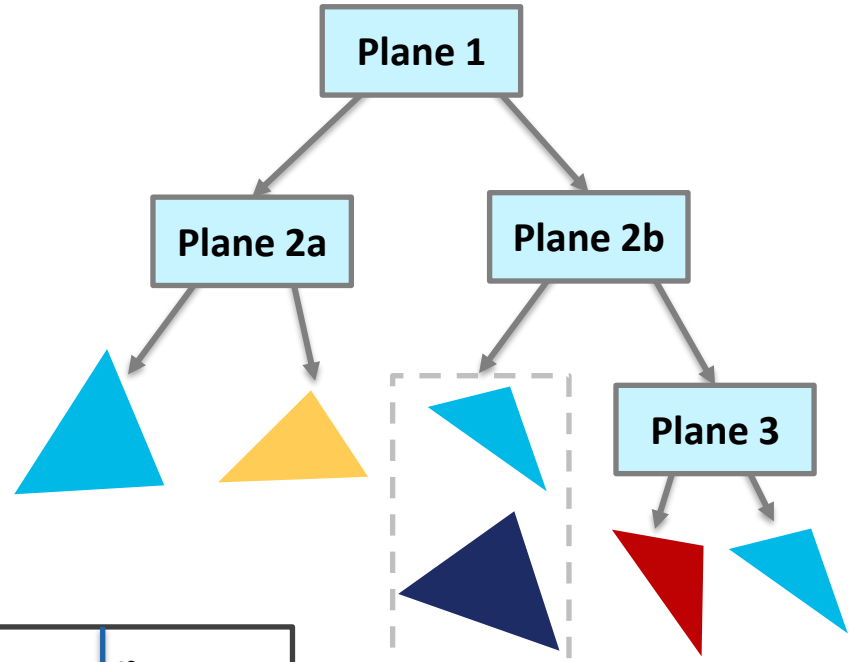
Enclose scene in axis-aligned bounding box (AABB)

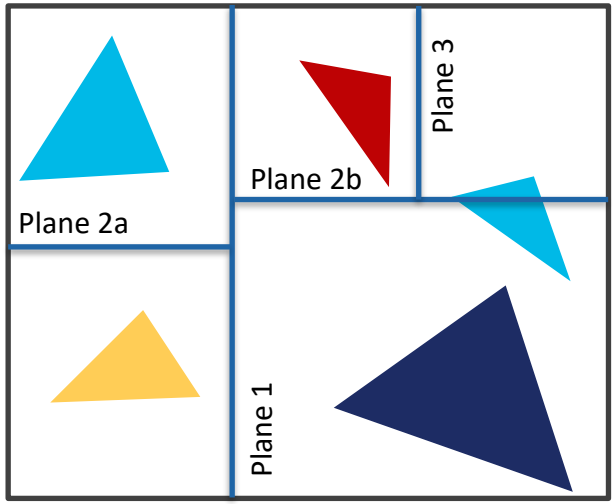
Recursively subdivide the AABB

- Choose axis and create a perpendicular plane creating two boxes
- Stop when some criterion is fulfilled (number of triangles, object size, …)

Can be used for rough front-to-back sorting

- Occlusion culling and reducing pixel overdraw





Scene

Polygon-aligned BSP Trees

Cut planes are aligned with polygons

- Tree can be traversed strictly from back to front (or front to back)

Used for visibility ordering of dynamic viewpoint

- BSP tree algorithm orders surfaces from back to front

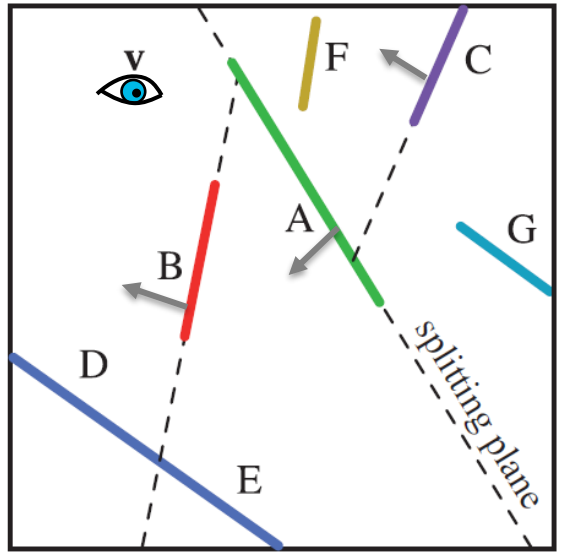
  Begin at root node

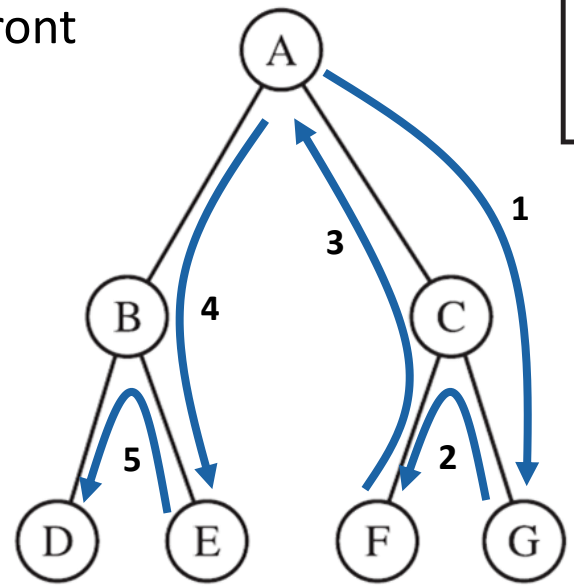
  Compare viewpoint with plane orientation

  Traverse subtree on other side first

  Select polygons in current node

  Traverse subtree on same side





G C F A E B D →

Guarantees correct occlusion order
but not that an object is closer

Octrees

Enclose scene in axis-aligned bounding box

Divide box recursively along all three axes

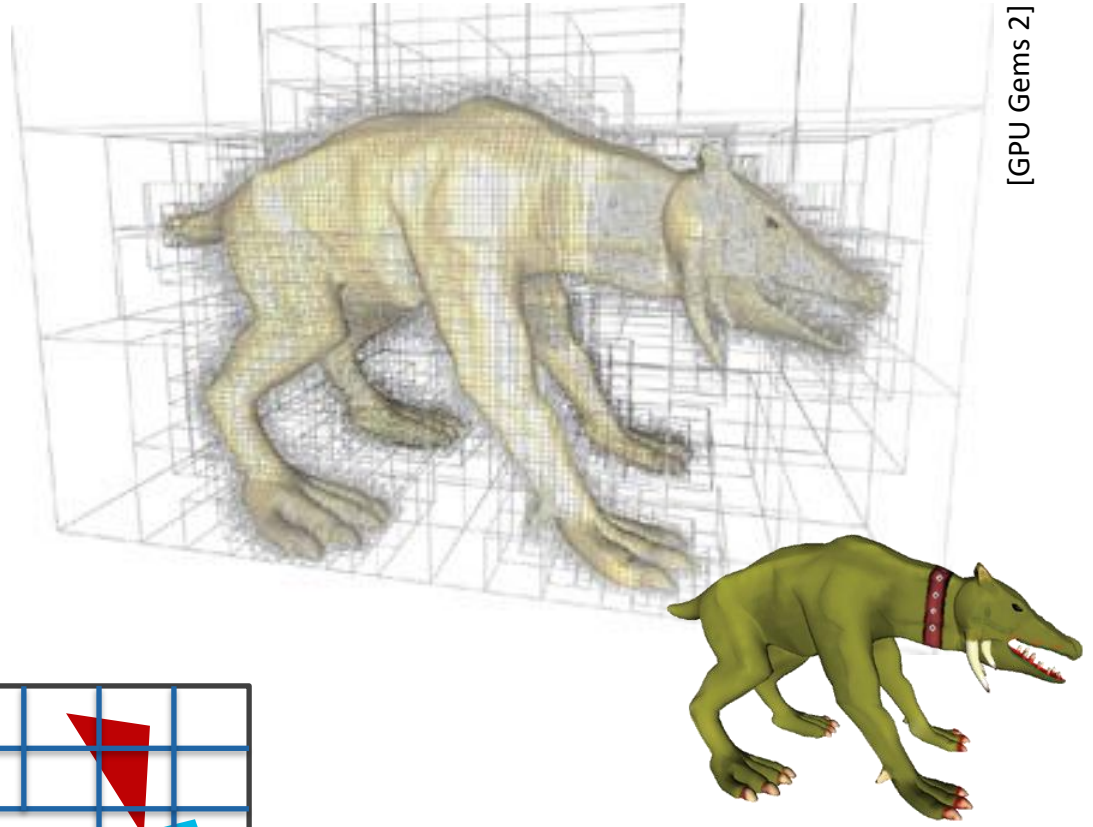
- Split point located at box center
- Creates 8 new boxes

Stop until each box is

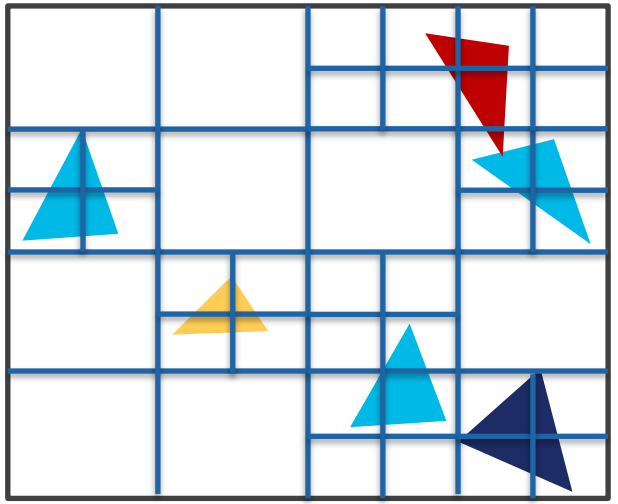
- Empty or
- Contains one object

2D equivalent: Quadtree

Similar to axis-aligned BSP tree



[GPU Gems 2]



Scene

Summary

– Real-time Graphics –

GPUs are massively parallel and optimized for rendering

- Graphics pipeline is programmable via shaders

Real-time rendering and shading

- Shading, texturing, and illumination per fragment
- Transparency
- Screen-space methods for ambient occlusion and deferred rendering

Data structures for large scenes

- Level-of-Detail (LoD)
- BVH, BSP, Octree
- Also applicable in offline rendering

References – Real-time Graphics

[Akenine-Möller 2018] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering*, Fourth Edition, CRC Press, 2018.

[Sun 2020] Yifan Sun, Nicolas Bohm Agostini, Shi Dong, and David Kaeli. *Summarizing CPU and GPU Design Trends with Product Data*. arXiv:1911.11313v2, 2020. https://arxiv.org/pdf/1911.11313.pdf

[GPU Gems] GPU Gems, Addison-Wesley Longman, 2004. https://developer.nvidia.com/gpugems/gpugems

[GPU Gems 2] GPU Gems 2, Addison-Wesley Longman, 2005. https://developer.nvidia.com/gpugems/gpugems2

Additional resources:

khronos.org	Standards for OpenGL, WebGL, Vulkan, OpenCL, …
realtimerendering.com/#intro	lists some freely available eBooks on various CG topics
thebookofshaders.com	Fragment shader guide
lighthouse3d.com	OpenGL & GLSL tutorials

Coming up next

Modeling

- Polygon meshes
- Parametric curves and surfaces
- Implicit modeling, CSG