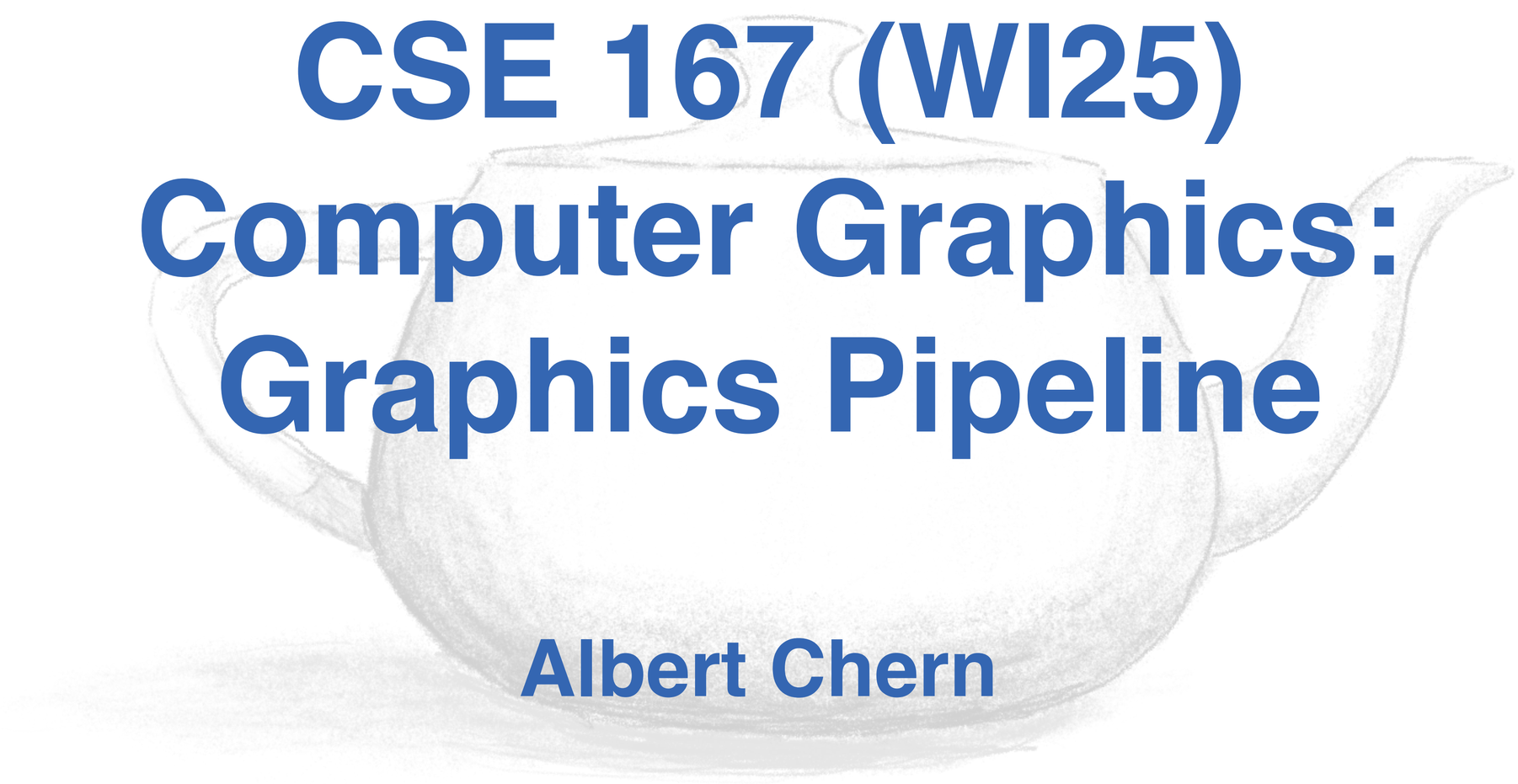
CSE 167 (WI25)
Computer Graphics:
Graphics Pipeline



Albert Chern

UC San Diego
JACOBS SCHOOL OF ENGINEERING

Overview

- An **algorithm for drawing picture**
- What is **rasterization**? What is **ray tracing**?
- What is **fragment**? What is **buffer**? What is **shader**?
- How do you **describe a shape**?
- What is **GPU**?

The Main Algorithm

- Main algorithm
- Rasterization pipeline
- GPU
- Summary

Big question

How to draw pictures algorithmically?

Big question

How to draw pictures algorithmically?

- Prescribe/input:

- Output:

Big question

How to draw pictures algorithmically?

- Prescribe/input:
  - Geometries (triangle mesh) in the scene
  - Color, material,… of each geometry
  - Light
  - Camera
- Output:
  - Color per pixel in the screen

Forza Motorsport 7
(Turn 10 Studios, Microsoft.)

Main algorithm

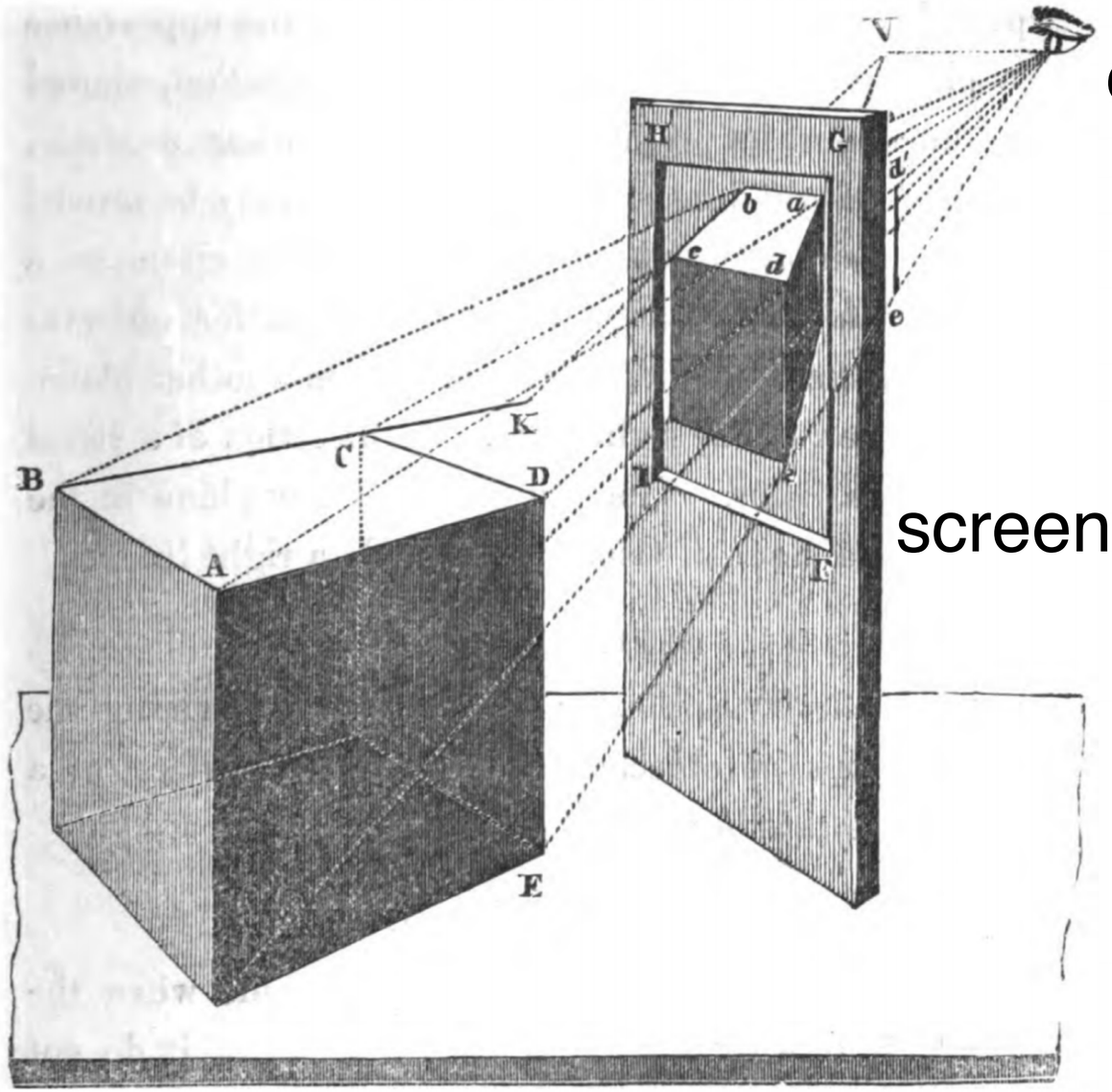
- Prescribe/input:
  - Geometries (triangle mesh) in the scene
  - Color, material,… of each geometry
  - Light
  - Camera

Step 1
Determine which triangle corresponds to which pixel

Step 2
Color each pixel according to lighting/material/orientation etc.

- Output:
  - Color per pixel in the screen

From 3D to 2D



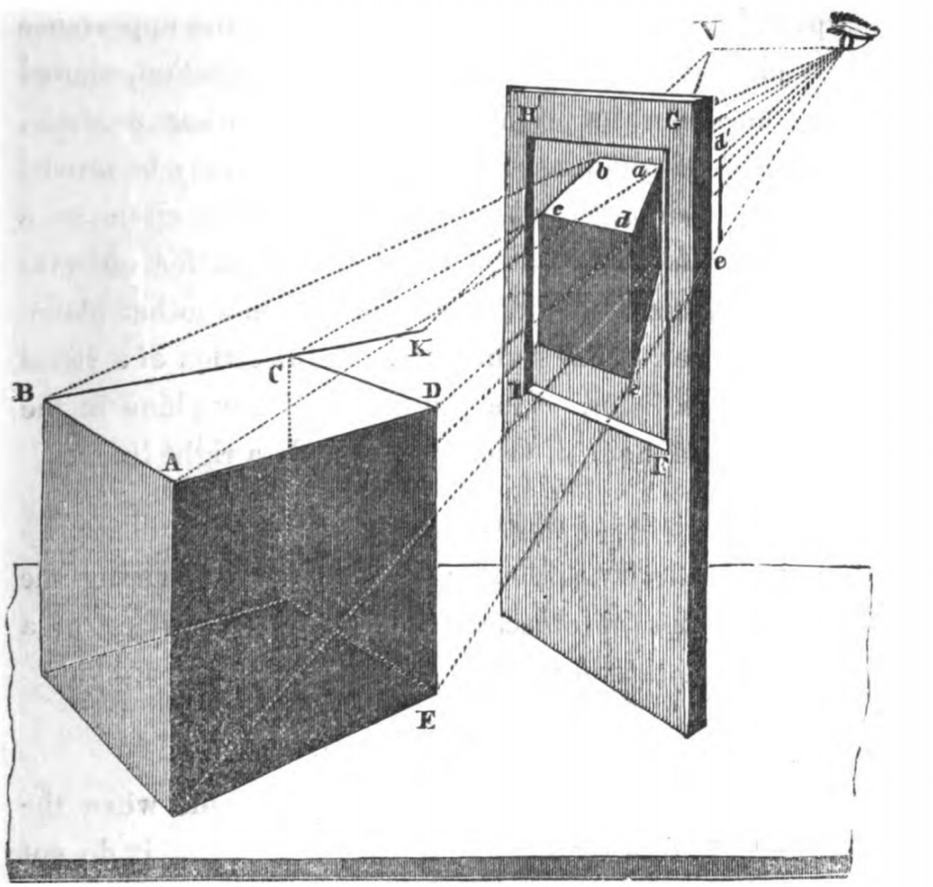
3D scene

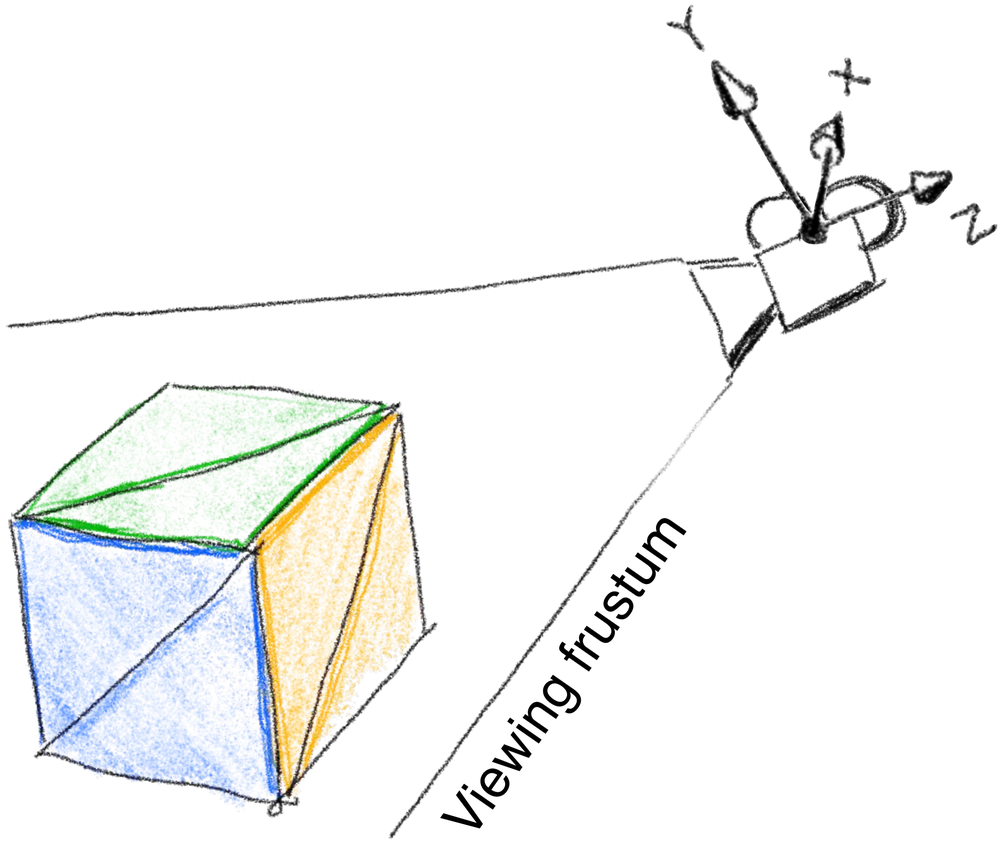
center of projection

screen

these are "camera"

From 3D to 2D

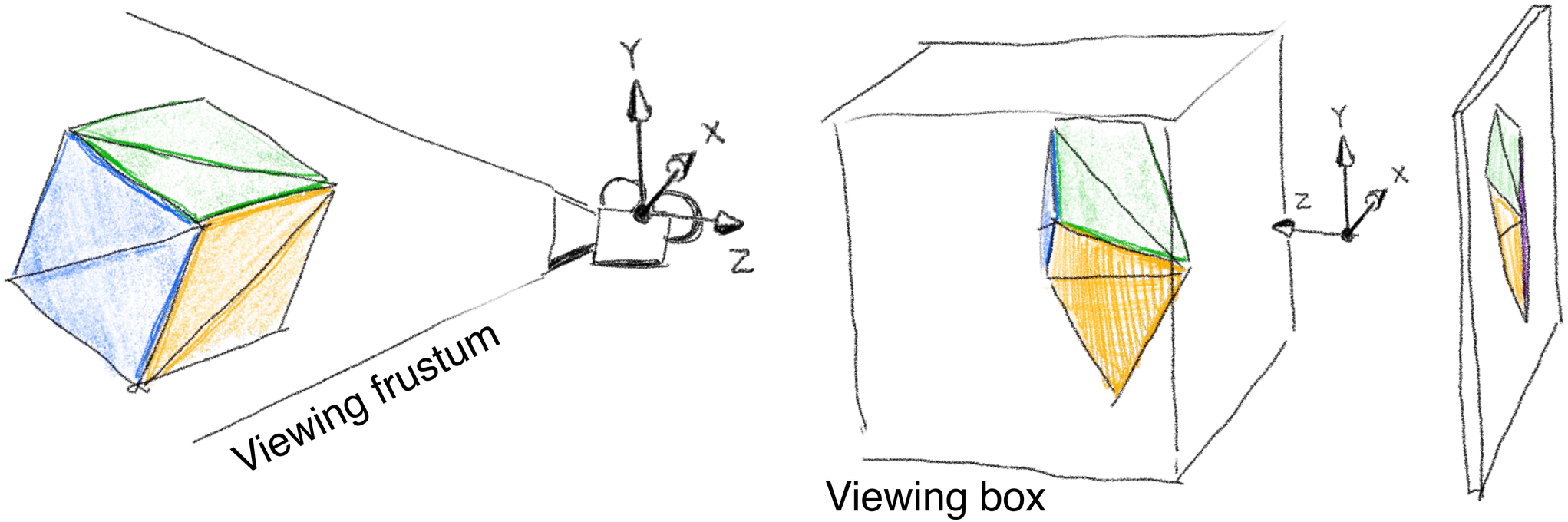




- Working out which geometries correspond to which pixels can be tedious

From 3D to 2D

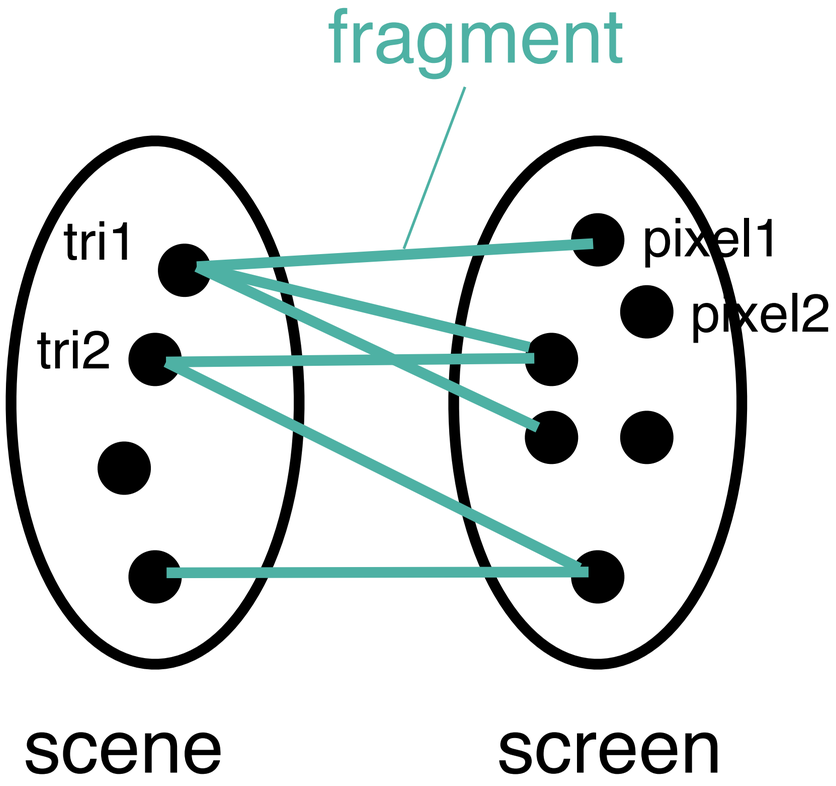
- In week 4, we will learn that there is always a transformation that normalizes the frustum into a standard box

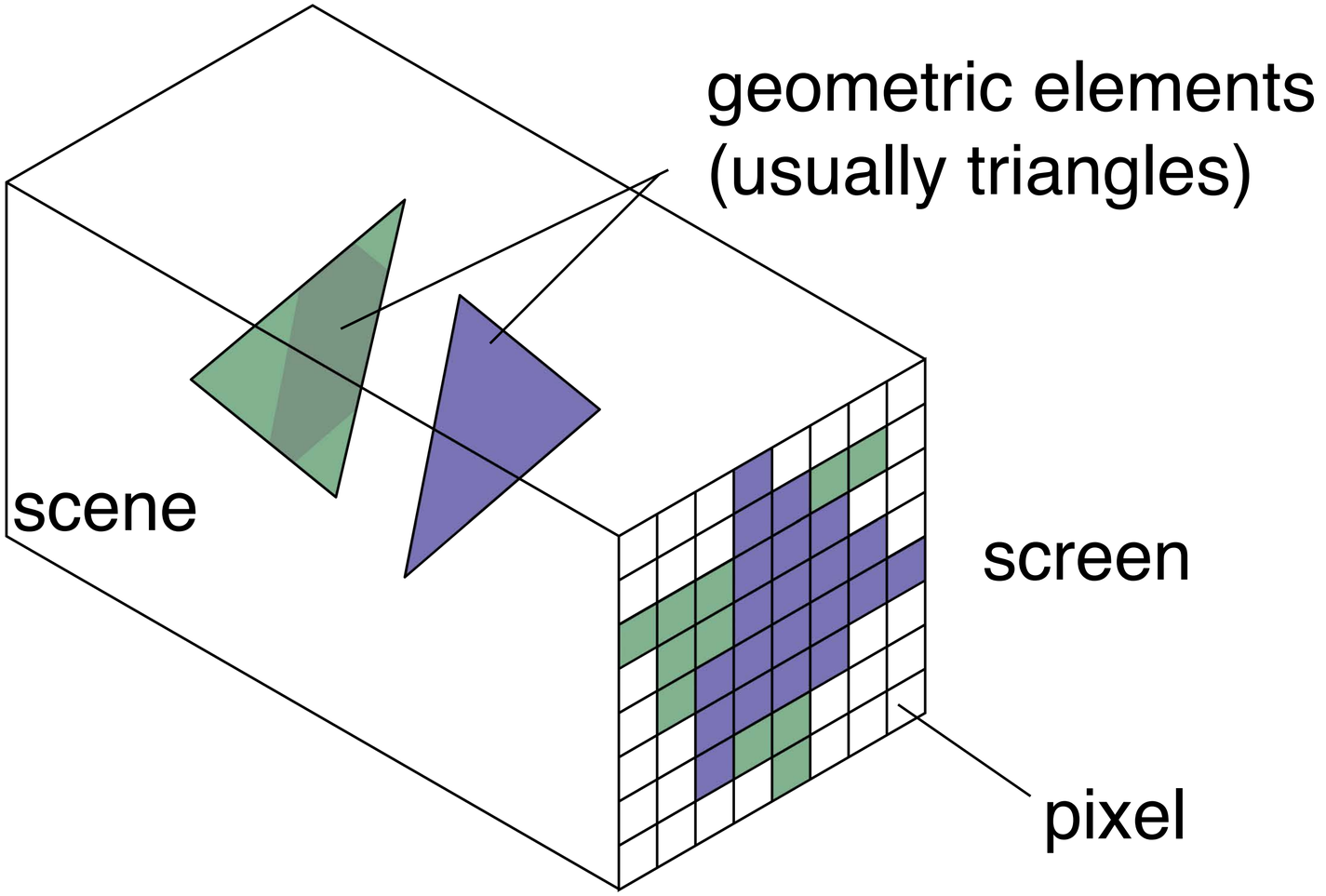


- The question remains. How do we turn triangles to pixels?

Scene-Screen Incidence Relation

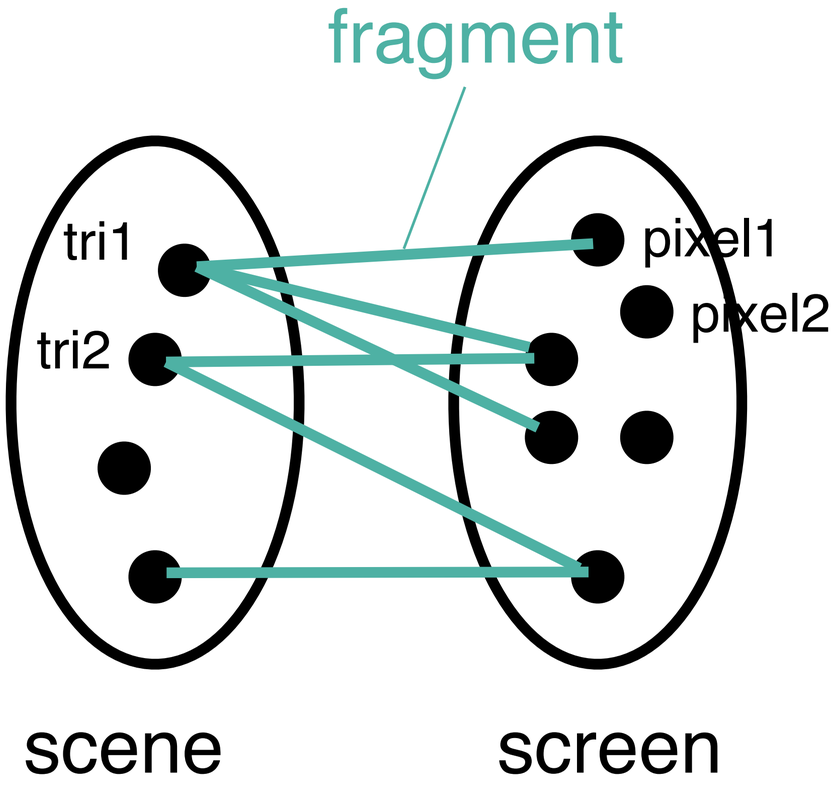
- Determine which triangles of the scene is incident to which pixels of the screen





Scene-Screen Incidence Relation

- Determine which triangles of the scene is incident to which pixels of the screen



Definition

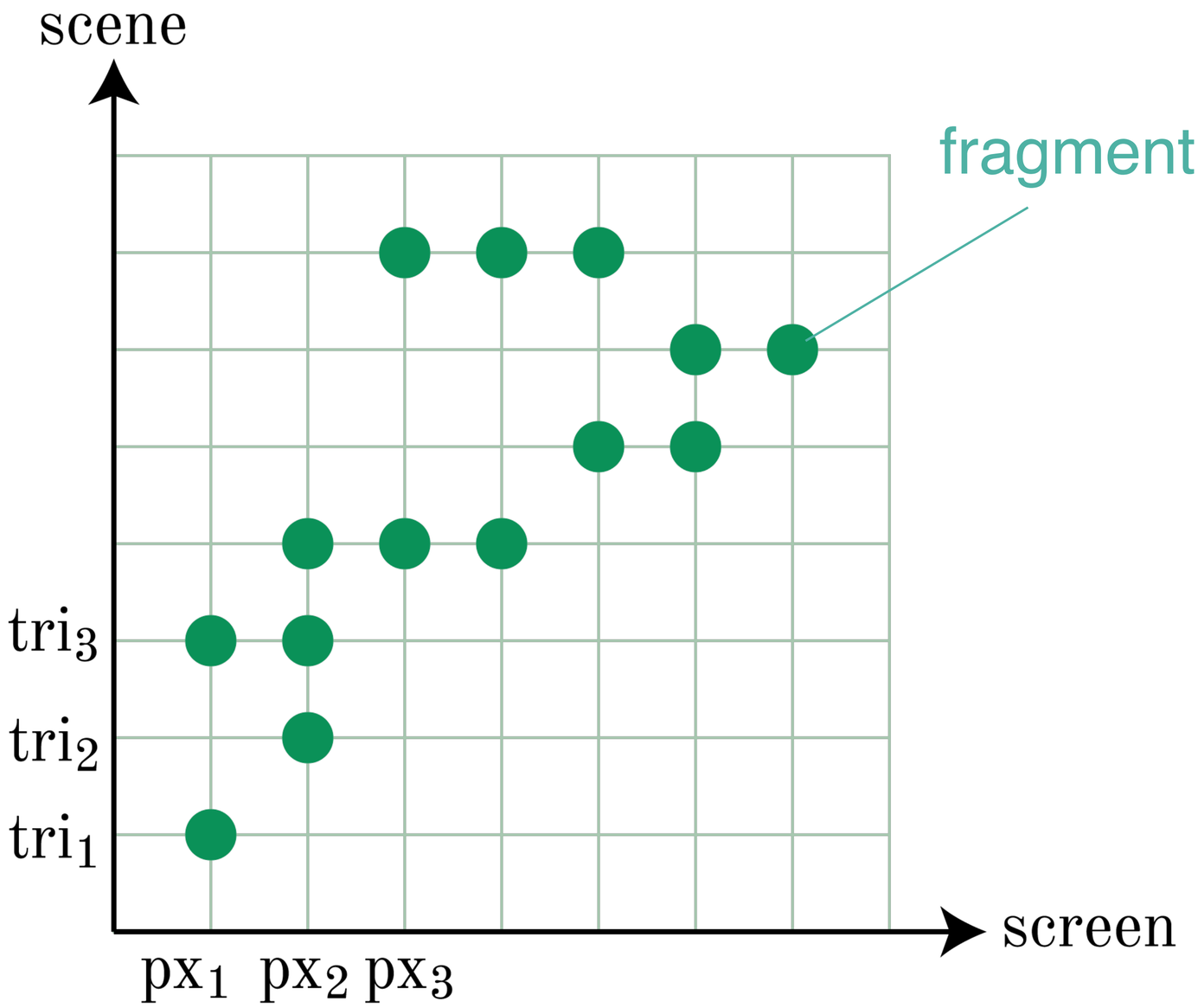
A ***fragment*** is a triangle-pixel pair so that the pixel is incident to (covered by) the triangle.

- Each fragment knows which triangle it comes from
- Each fragment knows which pixel to land on

Scene-Screen Incidence Relation

We are constructing a mapping between scene and screen.

There are 2 main ways for traversing and listing out all the fragments



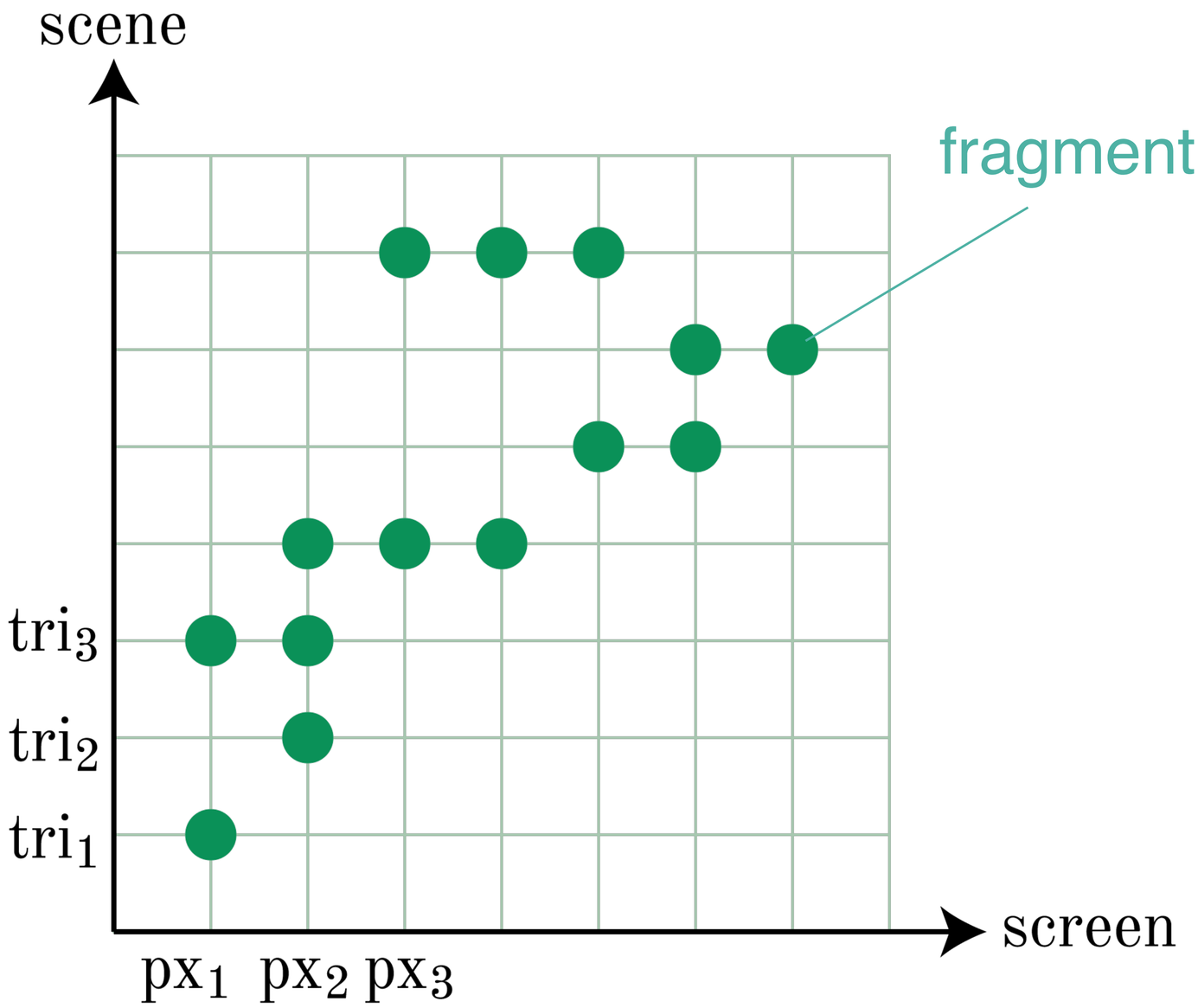
Scene-Screen Incidence Relation

Method 1

```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```

Method 2

```
for each pixel in screen
   for each triangle in scene
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```



Scene-Screen Incidence Relation

Method 1

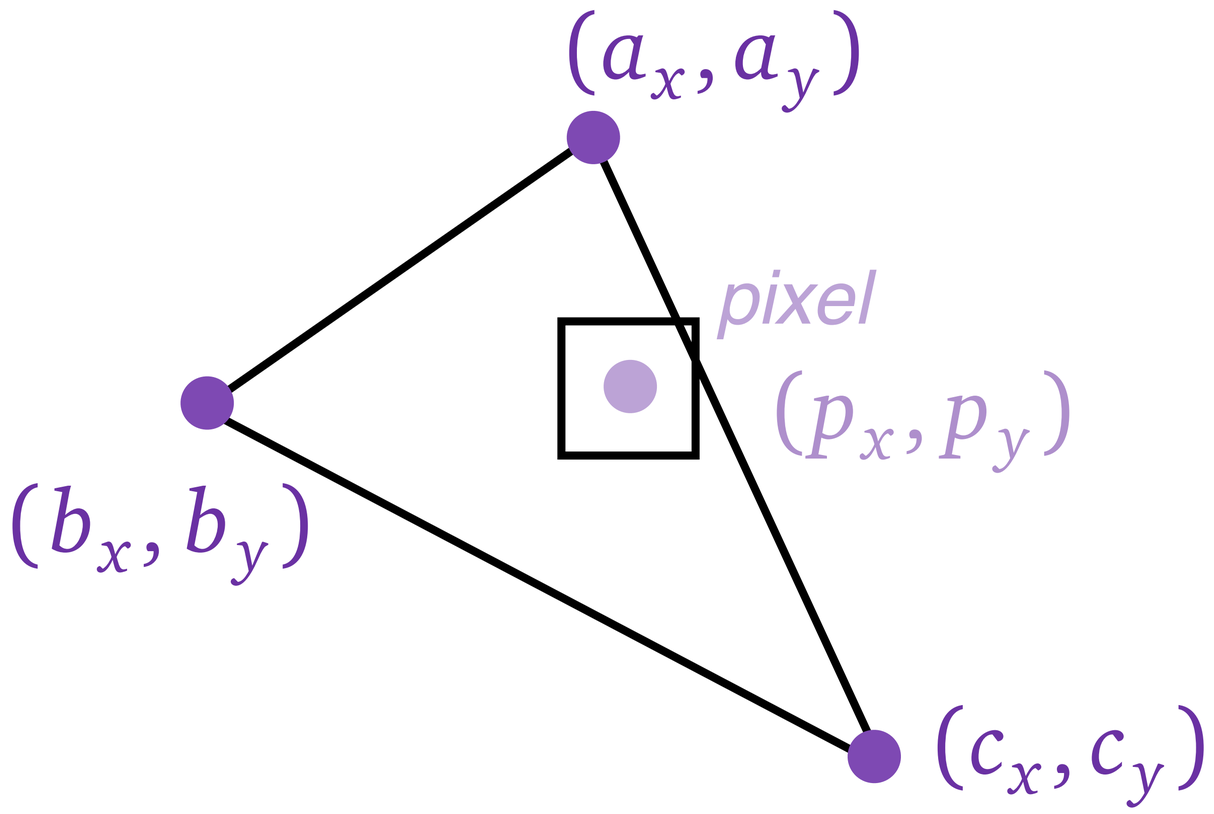
```
for each triangle in scene
  for each pixel in screen
    output the pair if the
      triangle occupies that
      pixel.
  end for
end for
```

Method 2

```
for each pixel in screen
  for each triangle in scene
    output the pair if the
      triangle occupies that
      pixel.
  end for
end for
```

We will learn this math problem in Week5:

What is the criterion that the pixel center is in the triangle?



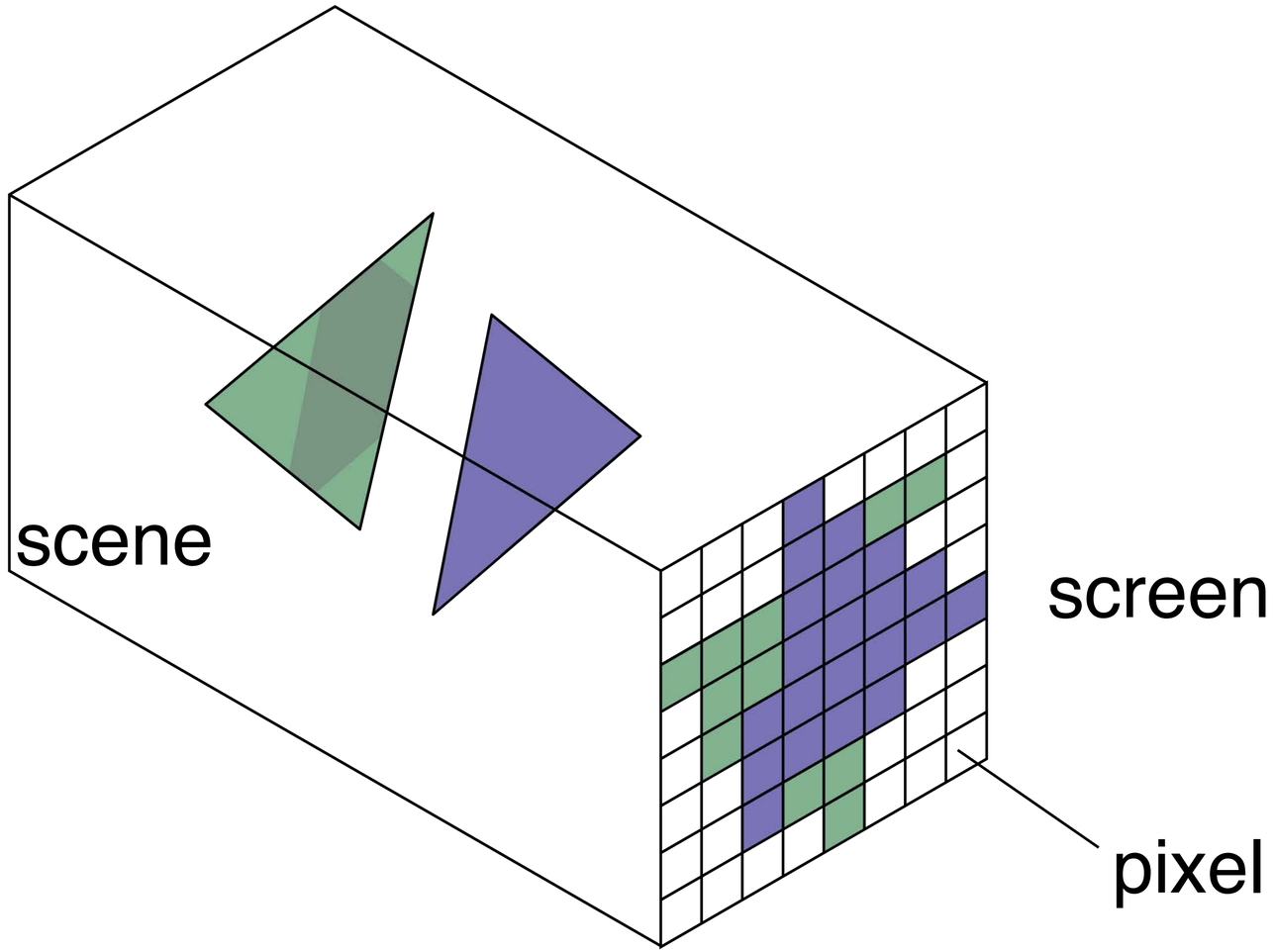
Scene-Screen Incidence Relation

Method 1

```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```

Method 2

```
for each pixel in screen
   for each triangle in scene
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```



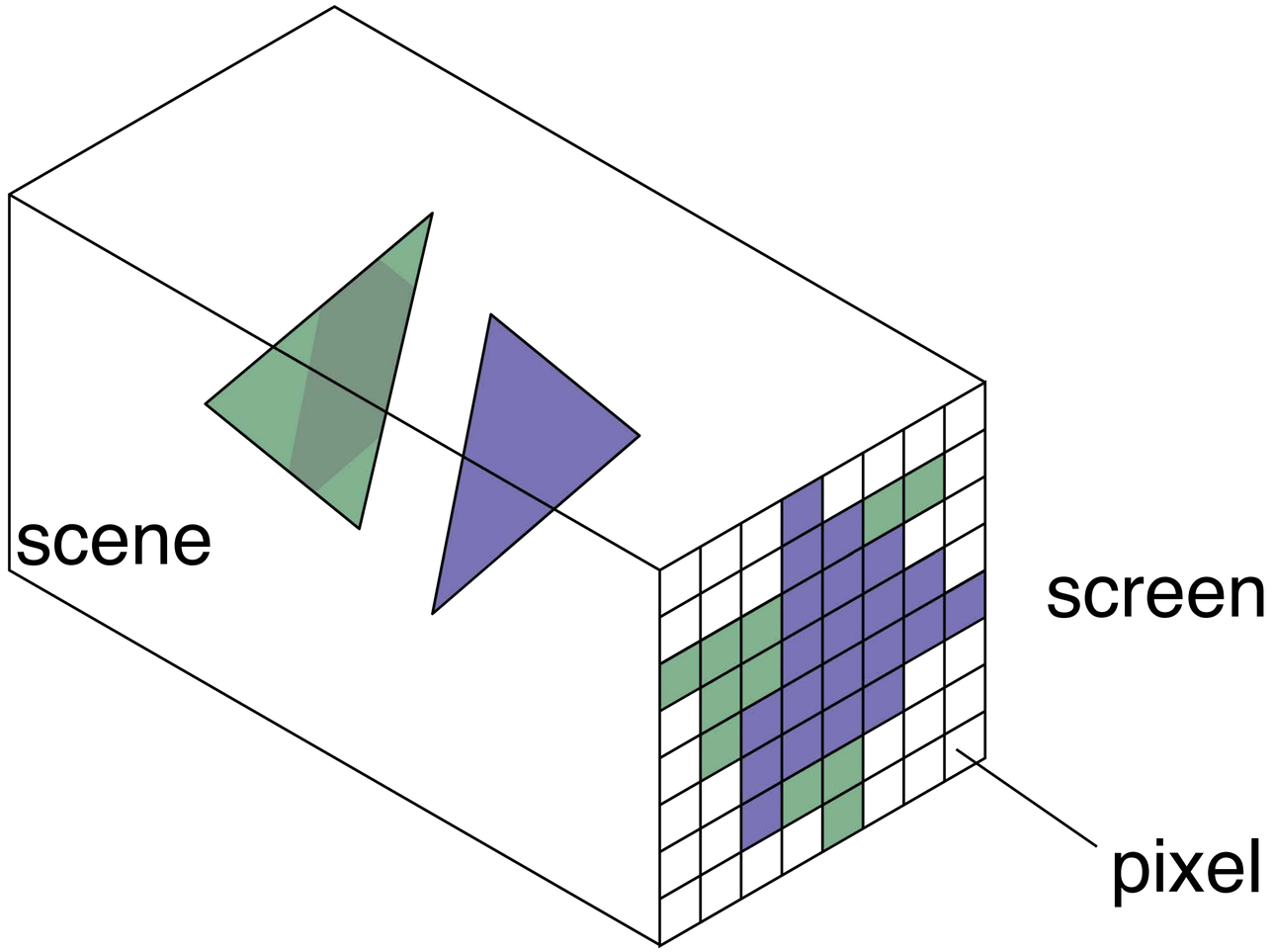
Scene-Screen Incidence Relation

Rasterization

```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```

Ray casting/tracing

```
for each pixel in screen
   for each triangle in scene
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```



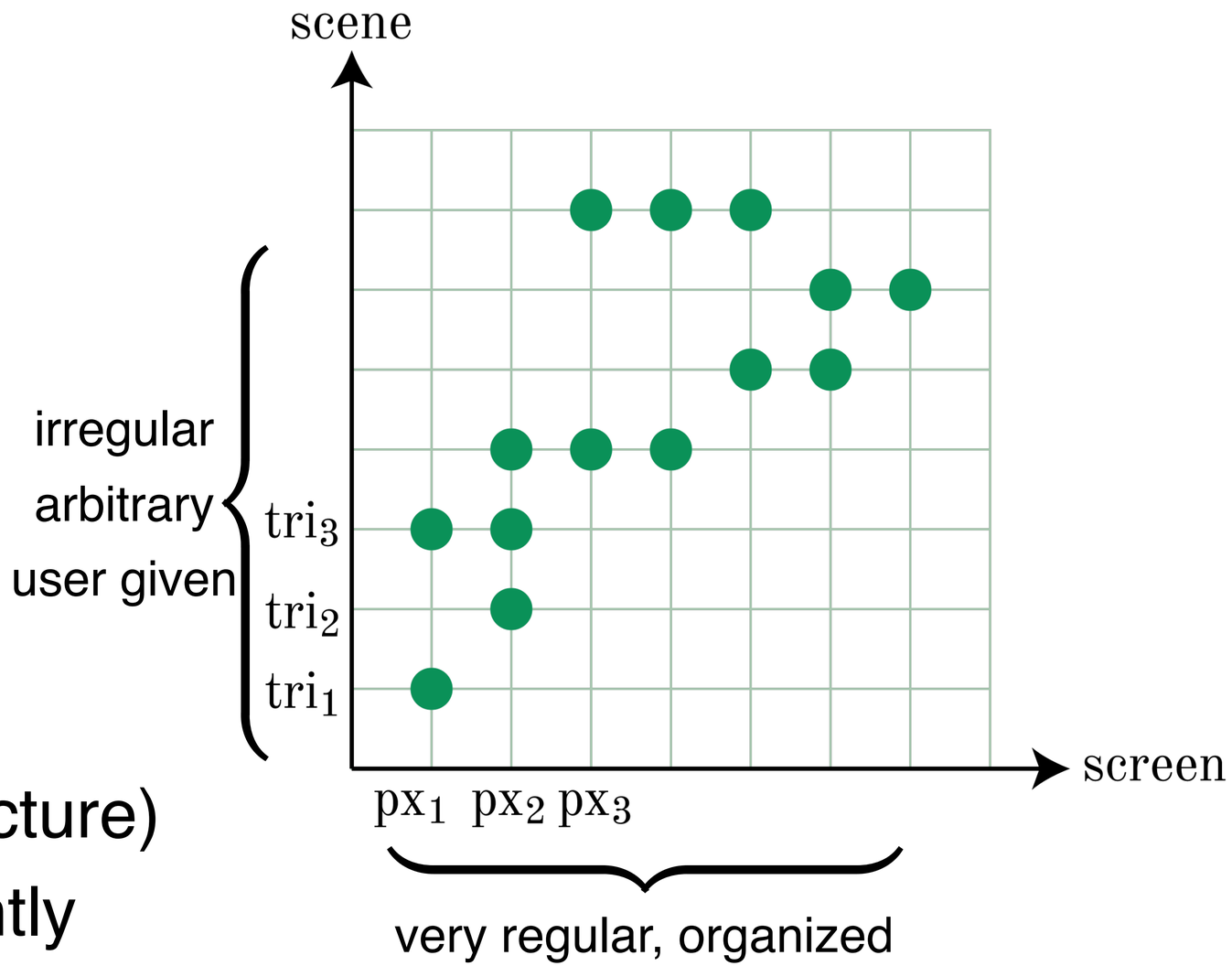
Rasterization v.s. Ray tracing

Rasterization

- Easy to speed up
- Hardcoded in hardware
- Real-time rendering
- No photorealism

Ray casting/tracing

- Hard to speed up
  (need nontrivial data structure)
- Hardware available recently
- Recursive ray tracing => photorealism



Rasterization v.s. Ray tracing

Rasterization

- Easy to speed up
- Hardcoded in hardware
- Real-time rendering
- No photorealism

Ray casting/tracing

- Hard to speed up
  (need nontrivial data structure)
- Hardware available recently
- Recursive ray tracing => photorealism

Rasterization v.s. Ray tracing

Rasterization

- Easy to speed up
- Hardcoded in hardware
- Real-time rendering
- No photorealism

First few weeks

- Runs on GPU
- OpenGL API

Ray casting/tracing

- Hard to speed up
  (need nontrivial data structure)
- Hardware available recently
- Recursive ray tracing => photorealism

Rasterization-based Graphics Pipeline

- Main algorithm
- Rasterization pipeline
- GPU
- Summary

Rasterization

Back to the main algorithm

Raster graphics pipeline

rasterizer

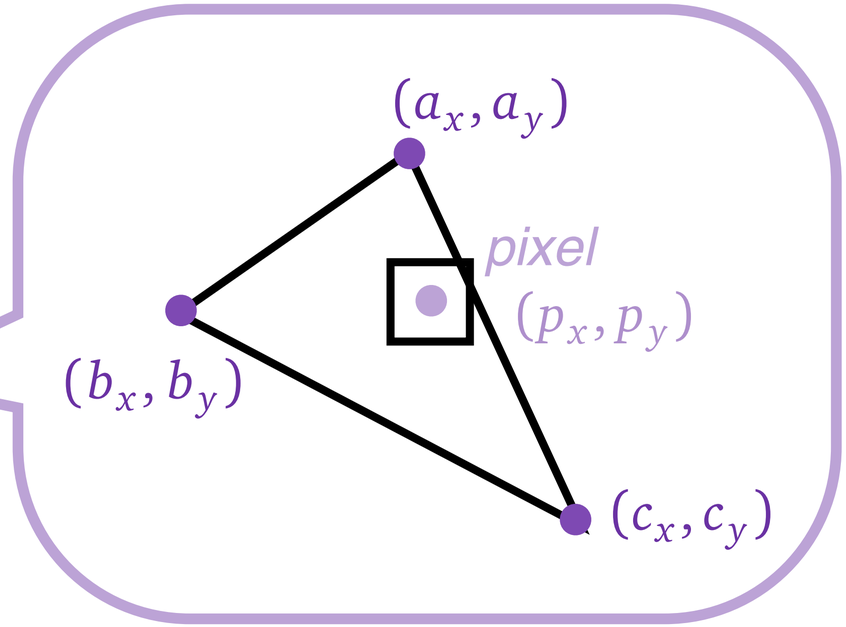
```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```

Raster graphics pipeline

rasterizer

Pseudocode

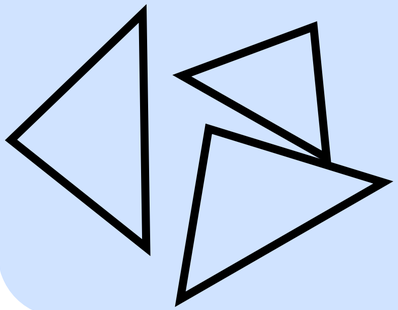
```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```



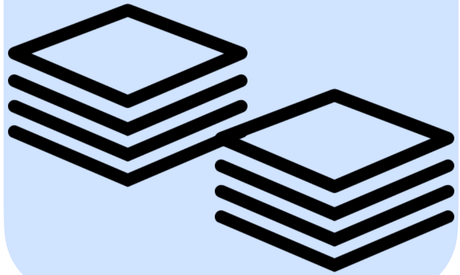
Data in allocated memory

a.k.a. **buffer**

list of triangles



list of fragments

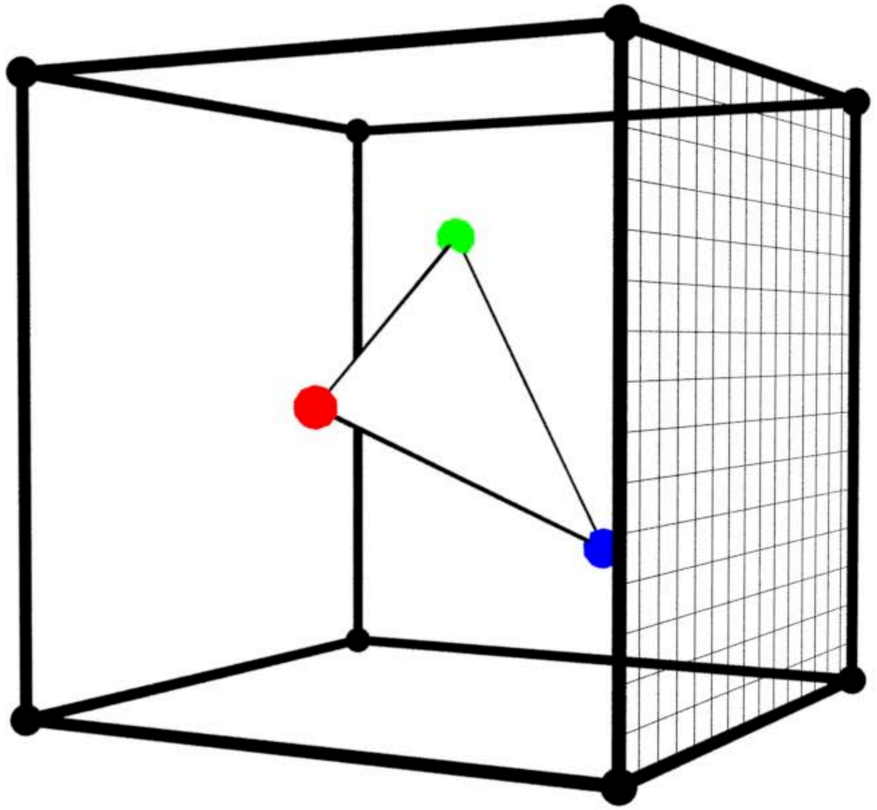


Raster graphics pipeline

rasterizer

```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```

Viewing box



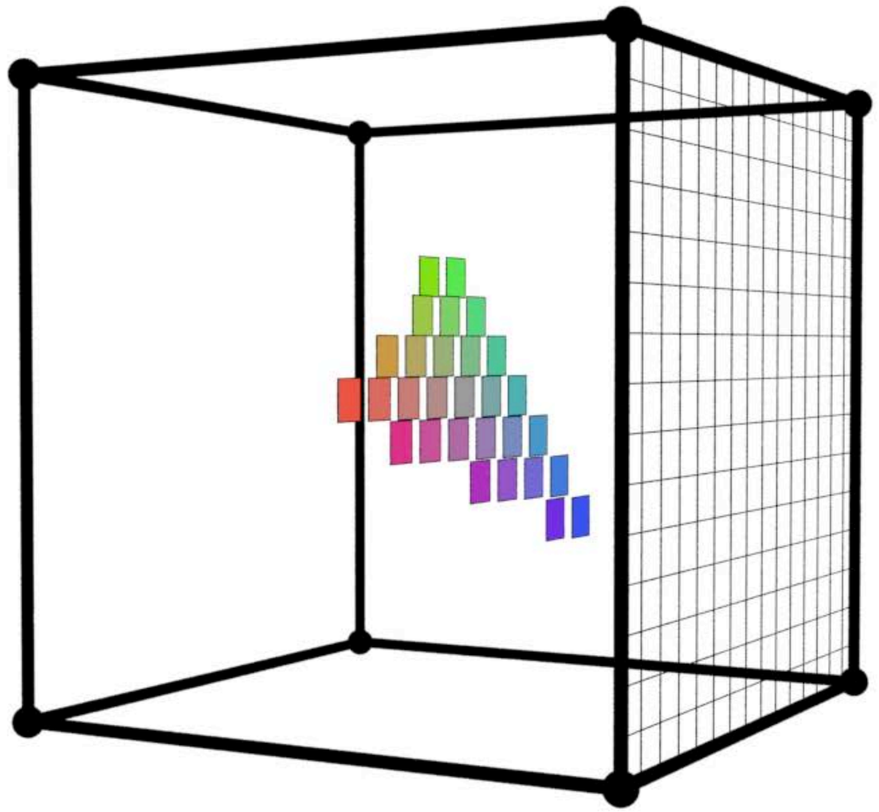
Screen

Raster graphics pipeline

rasterizer

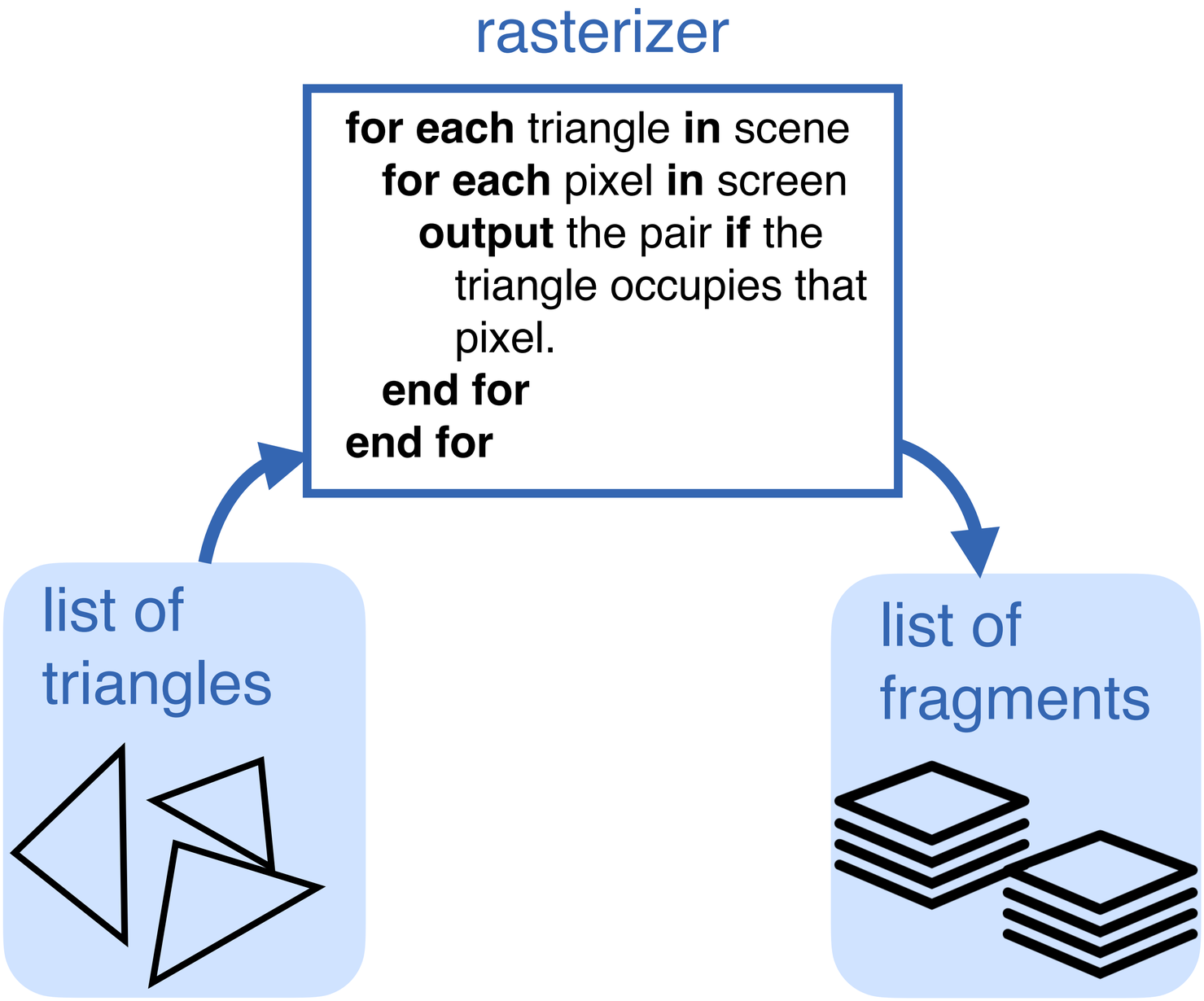
```
for each triangle in scene
   for each pixel in screen
      output the pair if the
         triangle occupies that
         pixel.
   end for
end for
```

Viewing box



Screen

Raster graphics pipeline



Raster graphics pipeline

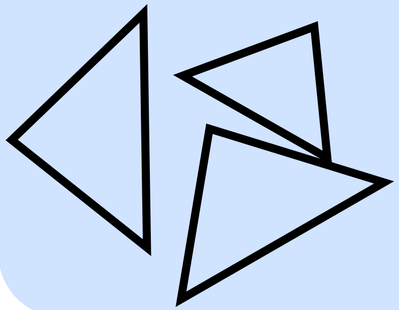
rasterizer

```
for each triangle in scene
  for each pixel in screen
    output the pair if the
      triangle occupies that
      pixel.
  end for
end for
```

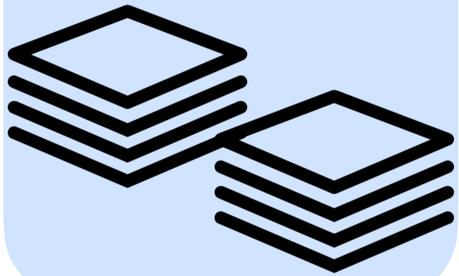
fragment shader

```
for each fragment
  …
  compute color
  …
  assign frag_color
end for
```

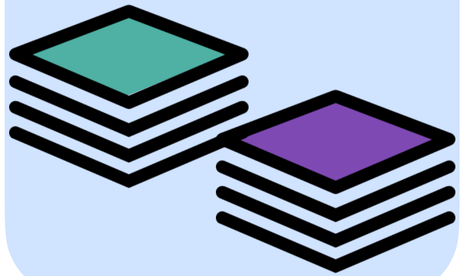
list of triangles



list of fragments



list of fragments



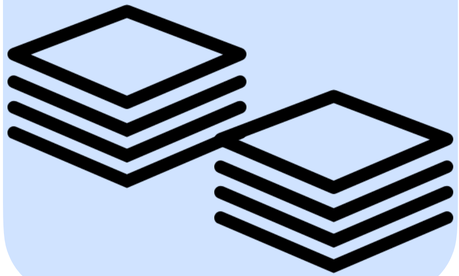
Raster graphics pipeline

rasterizer

```
for each triangle in scene
  for each pixel in screen
    output the pair if the
      triangle occupies that
      pixel.
  end for
end for
```

s

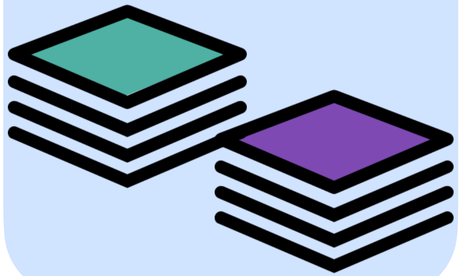
list of fragments



fragment shader

```
for each fragment
  …
  compute color
  …
  assign frag_color
end for
```

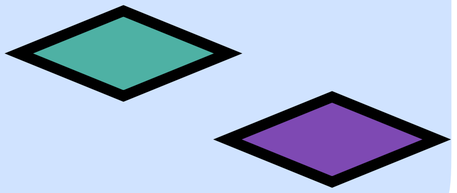
list of fragments



resolve depth

```
for each pixel
  keep only the
    top fragment
end for
```

color per pixel



Raster graphics pipeline

zer

e **in** scene
in screen
pair **if** the
ccupies that

fragment shader

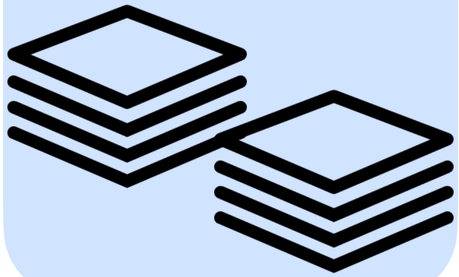
for each fragment
…
compute color
…
assign *frag_color*
end for

resolve depth

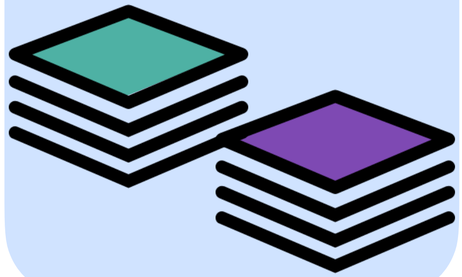
for each pixel
keep only the
top fragment
end for

Show on screen
or
Save as image

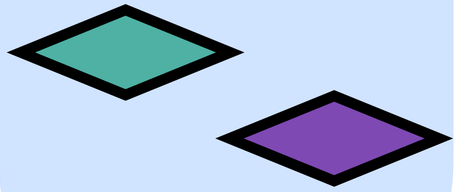
list of fragments



list of fragments



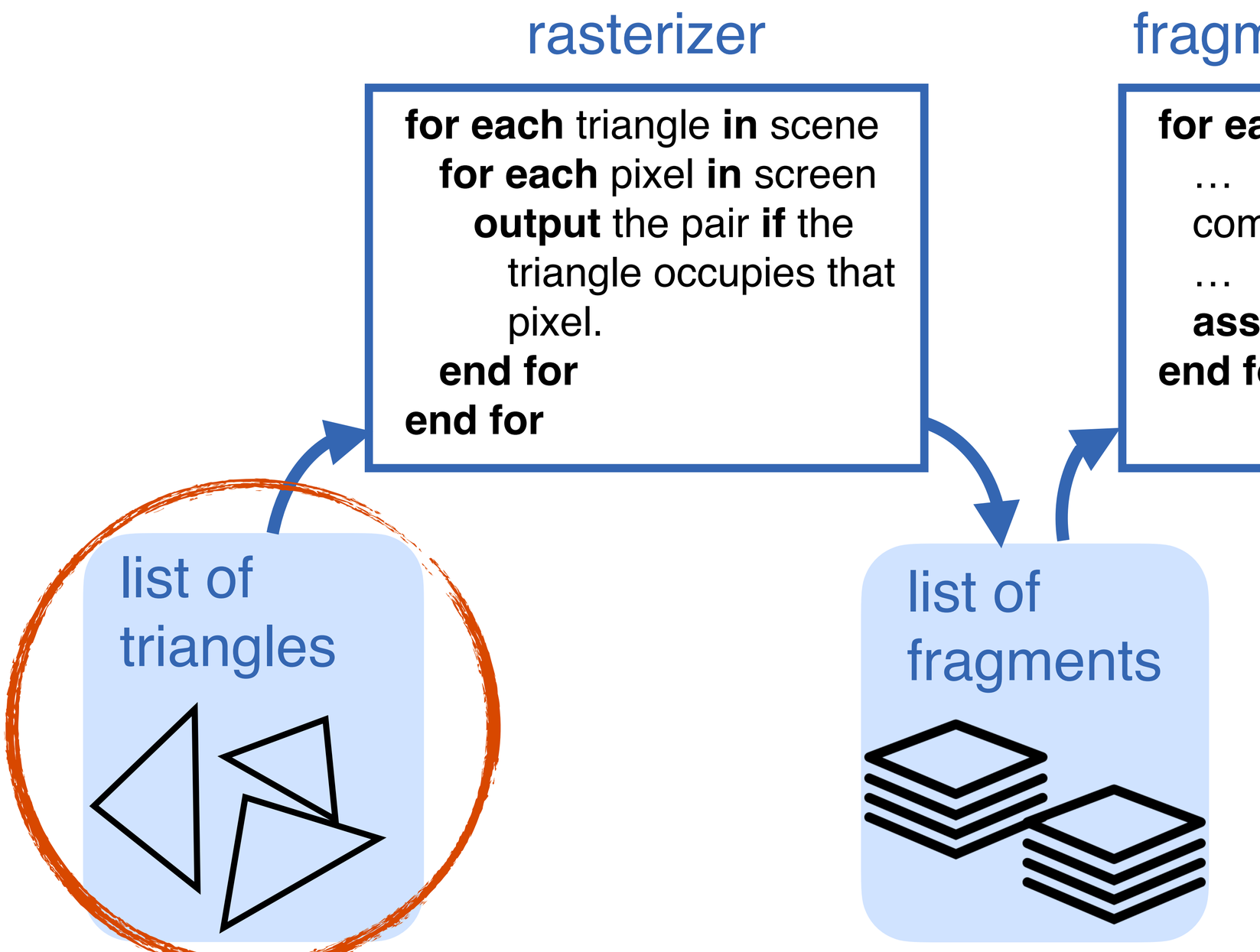
color per pixel



Frame buffer

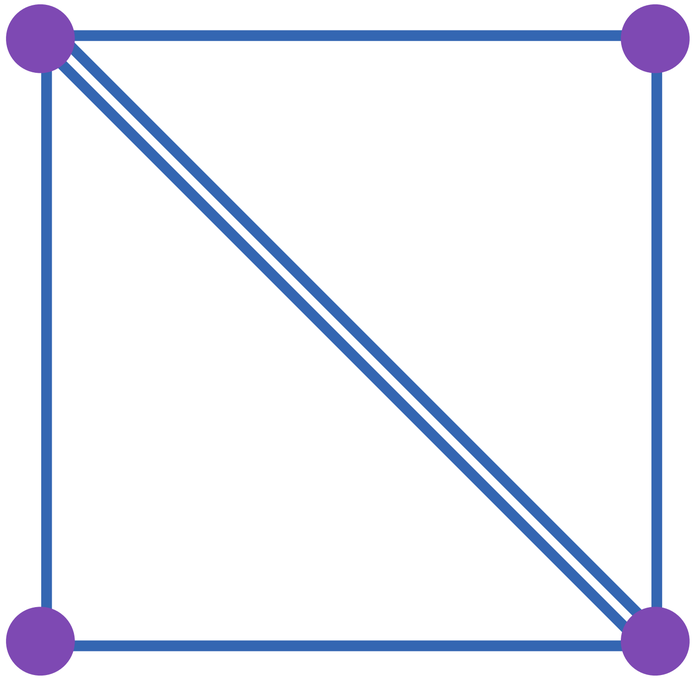
Raster graphics pipeline

How do we organize the data that describes a triangle mesh?



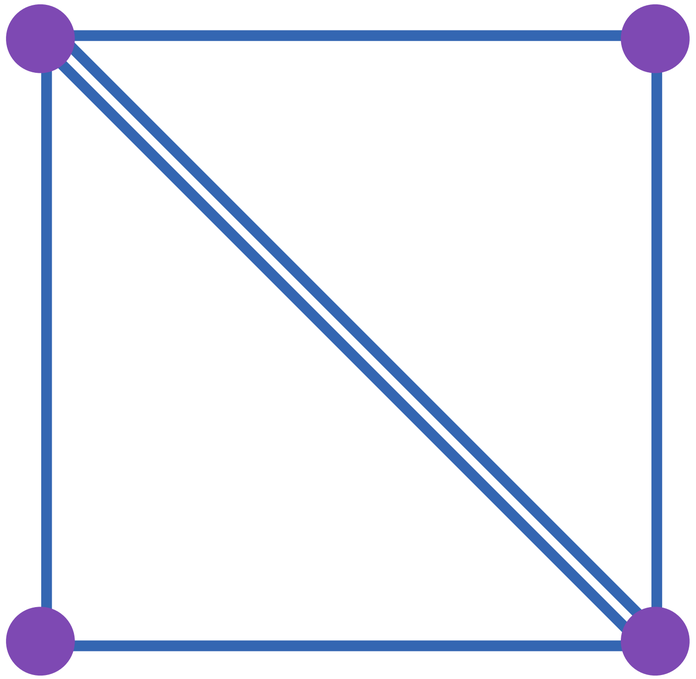
Description of triangle mesh

- Every shape (e.g. square) is partitioned into triangles mesh
- Triangle mesh consists of
  - Vertices
  - Connectivity



Description of triangle mesh

- Every shape (e.g. square) is partitioned into triangles mesh
- Triangle mesh consists of
  - Vertices
    - Position
    - Other attributes like color/material
  - Connectivity
    - Pointers to 3 vertices per triangle



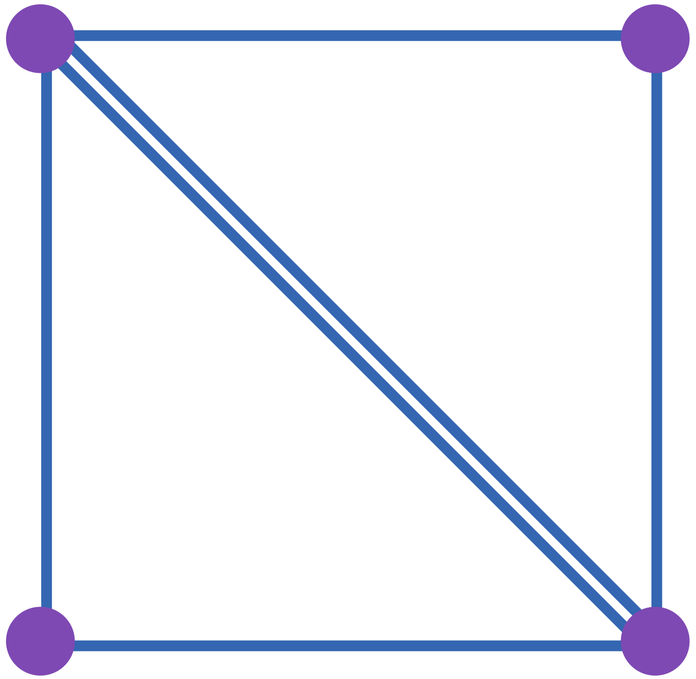
Description of triangle mesh

- Every shape (e.g. square) is partitioned into triangles mesh
- Triangle mesh consists of
  - Vertices

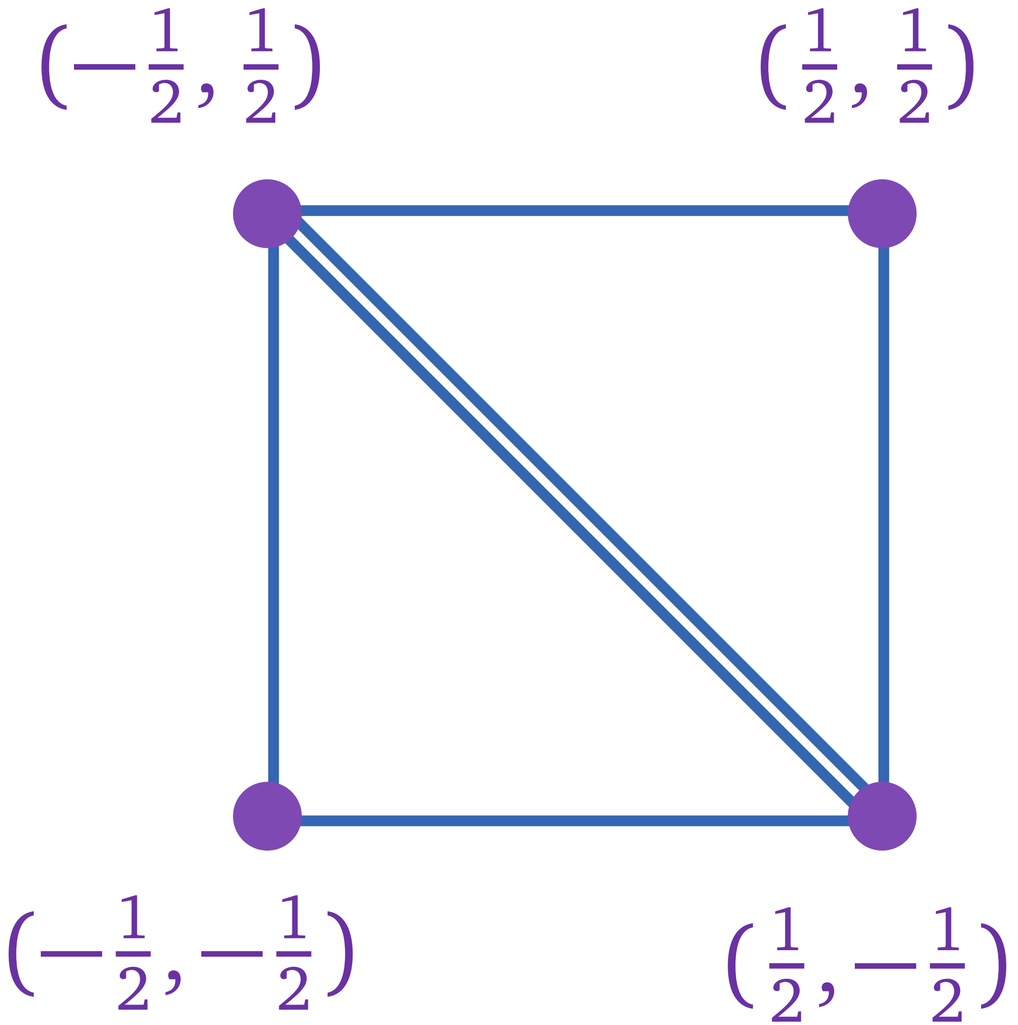
    we call the location storing these data the **vertex buffer**

    - Position
    - Other attributes like color/material
  - Connectivity
    - Pointers to 3 vertices per triangle

    we call the location storing these data the **index buffer**



Description of triangle mesh



Description of triangle mesh

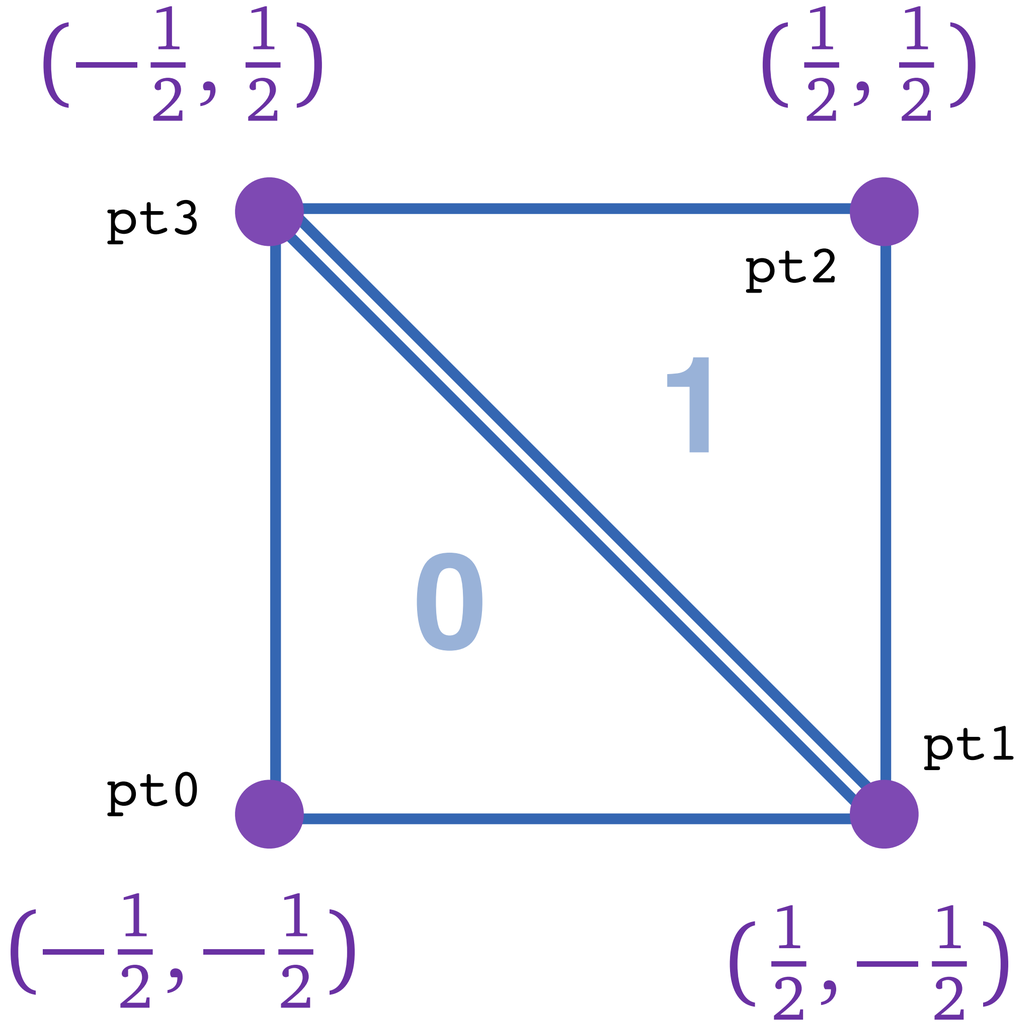
- Give an arbitrary order for vertices.
- Give an arbitrary order for triangles.

Vertex buffer

```
0   -0.5, -0.5,
1    0.5, -0.5,
2    0.5,  0.5,
3   -0.5,  0.5
```

Index buffer

```
0   0, 1, 3,
1   2, 3, 1
```



Description of triangle mesh

Every geometry is completely described in a *geometry spreadsheet* *(*and an instruction how to parse the vertex buffer)

Geometry spreadsheet
(a.k.a. **vertex array object (VAO)**)

Vertex buffer(s)

attribute

vertex

-0.5	-0.5	0.0	1.0	2.0	0.0
0.5	-0.5	0.0	1.0	1.0	1.0
0.5	0.5	0.0	1.0	1.5	-1.5
-0.5	0.5	0.0	1.0	1.0	1.0
0.0	0.0	1.0	1.0	0.5	0.5
0.0	0.0	-1.0	1.0	0.5	0.5
-0.5	-0.5	0.0	1.0	2.0	0.0
0.5	-0.5	0.0	1.0	1.0	1.0
0.5	0.5	0.0	1.0	1.5	-1.5
-0.5	0.5	0.0	1.0	1.0	1.0
0.0	0.0	1.0	1.0	0.5	0.5
0.0	0.0	-1.0	1.0	0.5	0.5

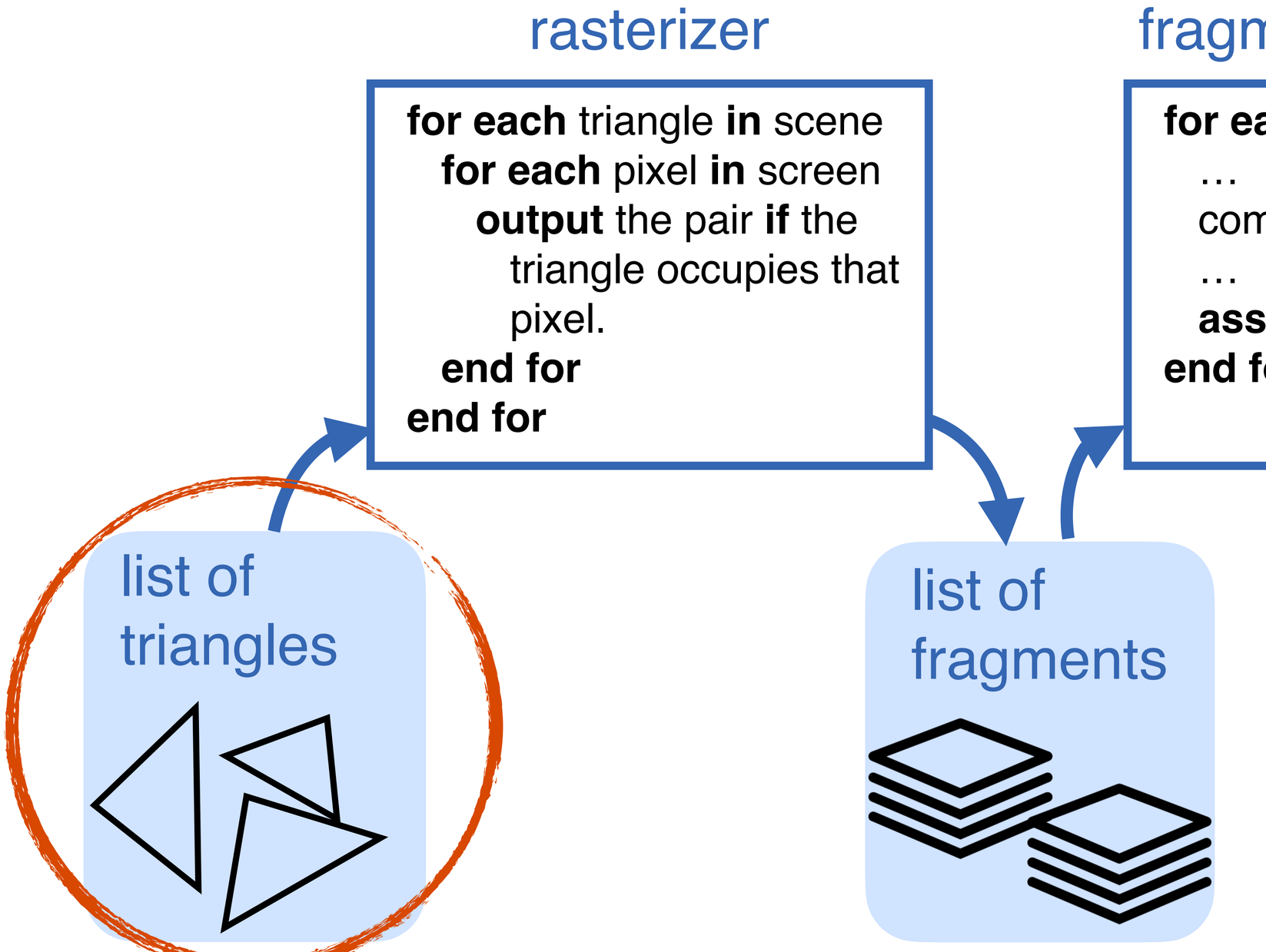
Index buffer

vertex

triangle

0	1	3
2	3	1
3	4	6
5	6	4
10	12	7
8	9	11
10	11	13
12	13	11
13	14	16
15	16	14
20	22	17
18	19	21

Raster graphics pipeline



Raster graphics pipeline

rasterizer

```
for each triangle in scene
  for each pixel in screen
    output the pair if the
      triangle occupies that
      pixel.
  end for
end for
```

frag

```
for ea
  …
  com
  …
  ass
end f
```

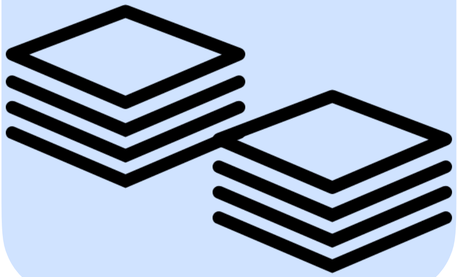
vertex buffer

-0.5	-0.5	0.0	1.0	2.0	0.0
0.5	-0.5	0.0	1.0	1.0	1.0
0.5	0.5	0.0	1.0	1.5	-1.5
-0.5	0.5	0.0	1.0	1.0	1.0
0.0	0.0	1.0	1.0	0.5	0.5
0.0	0.0	-1.0	1.0	0.5	0.5
-0.5	-0.5	0.0	1.0	2.0	0.0
0.5	-0.5	0.0	1.0	1.0	1.0
0.5	0.5	0.0	1.0	1.5	-1.5
-0.5	0.5	0.0	1.0	1.0	1.0

index buffer

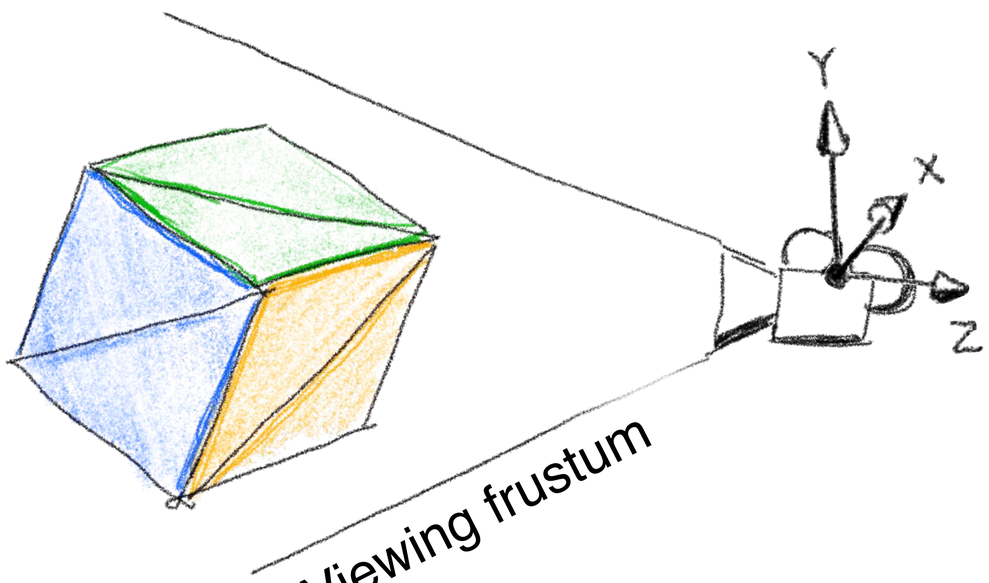
0	1	3
2	3	1
3	4	6
5	6	4
10	12	7
8	9	11
10	11	13
12	13	11
13	14	16
15	16	14

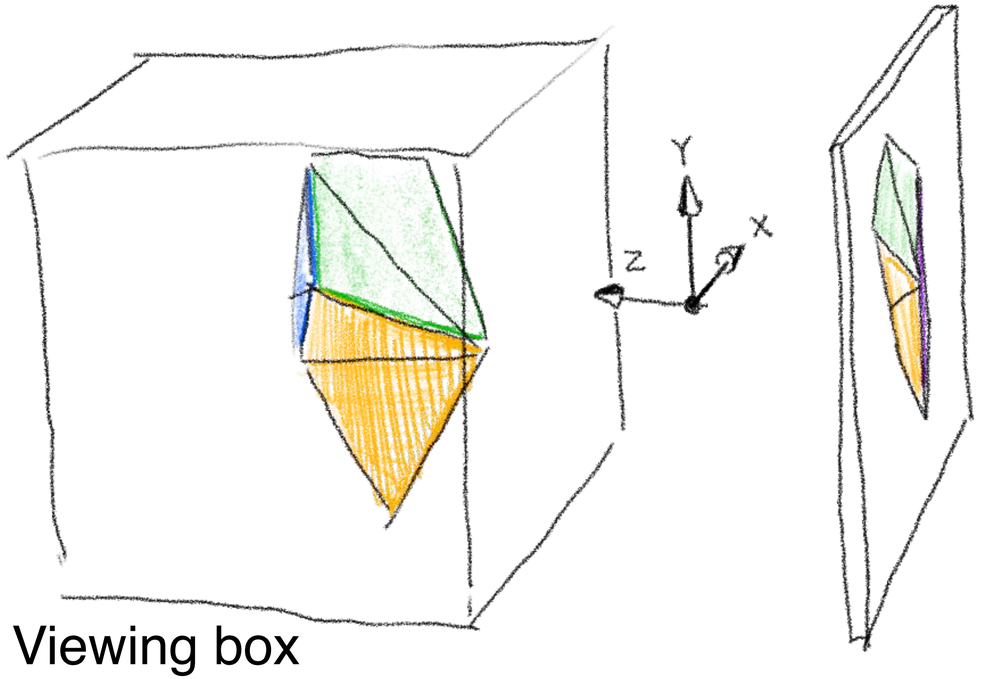
list of fragments



Raster graphics pipeline

Recall that before the rasterization, we need to perform some transformation…





Raster graphics pipeline

vertex shader

for each vertex
  parse vert. attrib.
  compute transform
  …
  output *coordinate in the viewing box*
end for

rasterizer

for each triangle **in** scene
  for each pixel **in** screen
    output the pair **if** the triangle occupies that pixel.
  end for
end for

fragm

for ea
  …
  com
  …
  ass
end f

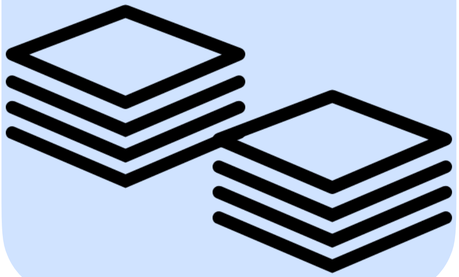
vertex buffer

-0.5	-0.5	0.0	1.0	2.0	0.0
0.5	-0.5	0.0	1.0	1.0	1.0
0.5	0.5	0.0	1.0	1.5	-1.5
-0.5	0.5	0.0	1.0	1.0	1.0
0.0	0.0	1.0	1.0	0.5	0.5
0.0	0.0	-1.0	1.0	0.5	0.5
-0.5	-0.5	0.0	1.0	2.0	0.0
0.5	-0.5	0.0	1.0	1.0	1.0
0.5	0.5	0.0	1.0	1.5	-1.5
-0.5	0.5	0.0	1.0	1.0	1.0

index buffer

0	1	3
2	3	1
3	4	6
5	6	4
10	12	7
8	9	11
10	11	13
12	13	11
13	14	16
15	16	14

list of fragments



Raster graphics pipeline

vertex shader

for each vertex
  parse vert. attrib.
  compute transform
  …
  output *coordinate in the viewing box*
end for

rasterizer

for each triangle **in** scene
  for each pixel **in** screen
    output the pair **if** the triangle occupies that pixel.
  end for
end for

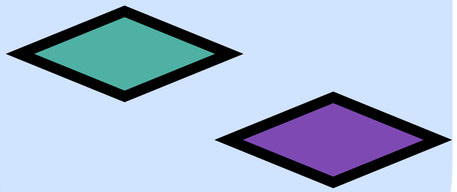
fragment shader

for each fragment
  …
  compute color
  …
  assign *frag_color*
end for

resolve depth

for each pixel
  keep only the top fragment
end for

color per pixel



Raster graphics pipeline

vertex shader

```
for each vertex
    parse vert. attrib.
    compute transform
    …
    output coordinate
     in the viewing box
end for
```

flexible

rasterizer

```
for each triangle in scene
    for each pixel in screen
        output the pair if the
            triangle occupies that
            pixel.
    end for
end for
```

fixed, hardcoded

fragment shader

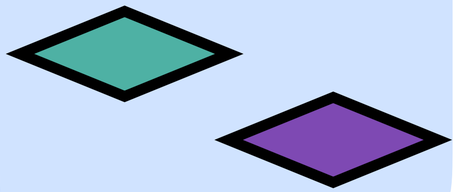
```
for each fragment
    …
    compute color
    …
    assign frag_color
end for
```

flexible

resolve depth

```
for each pixel
    keep only the
        top fragment
end for
```

color per pixel



- Flexible: Frequently reprogrammed depending on application
- Fixed: Hardcode in a chip
- Every **for each** loop can be parallelized

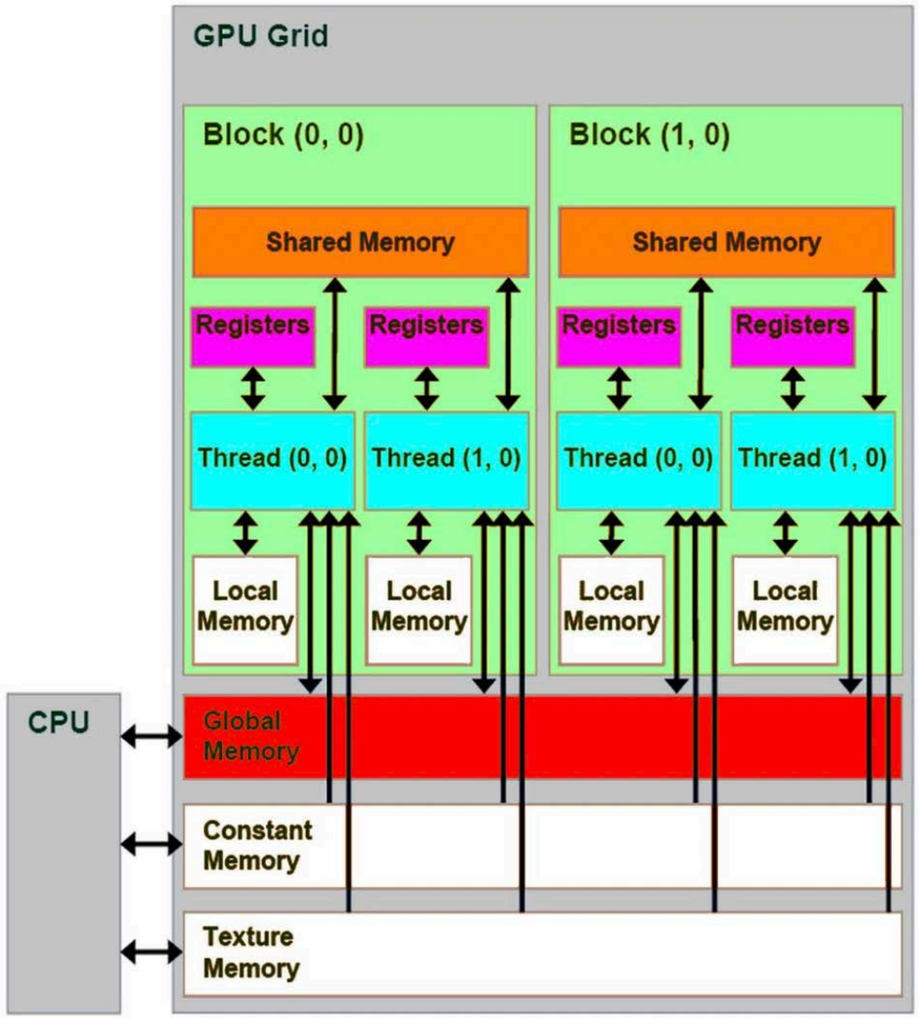
Graphics Processing Unit (GPU)

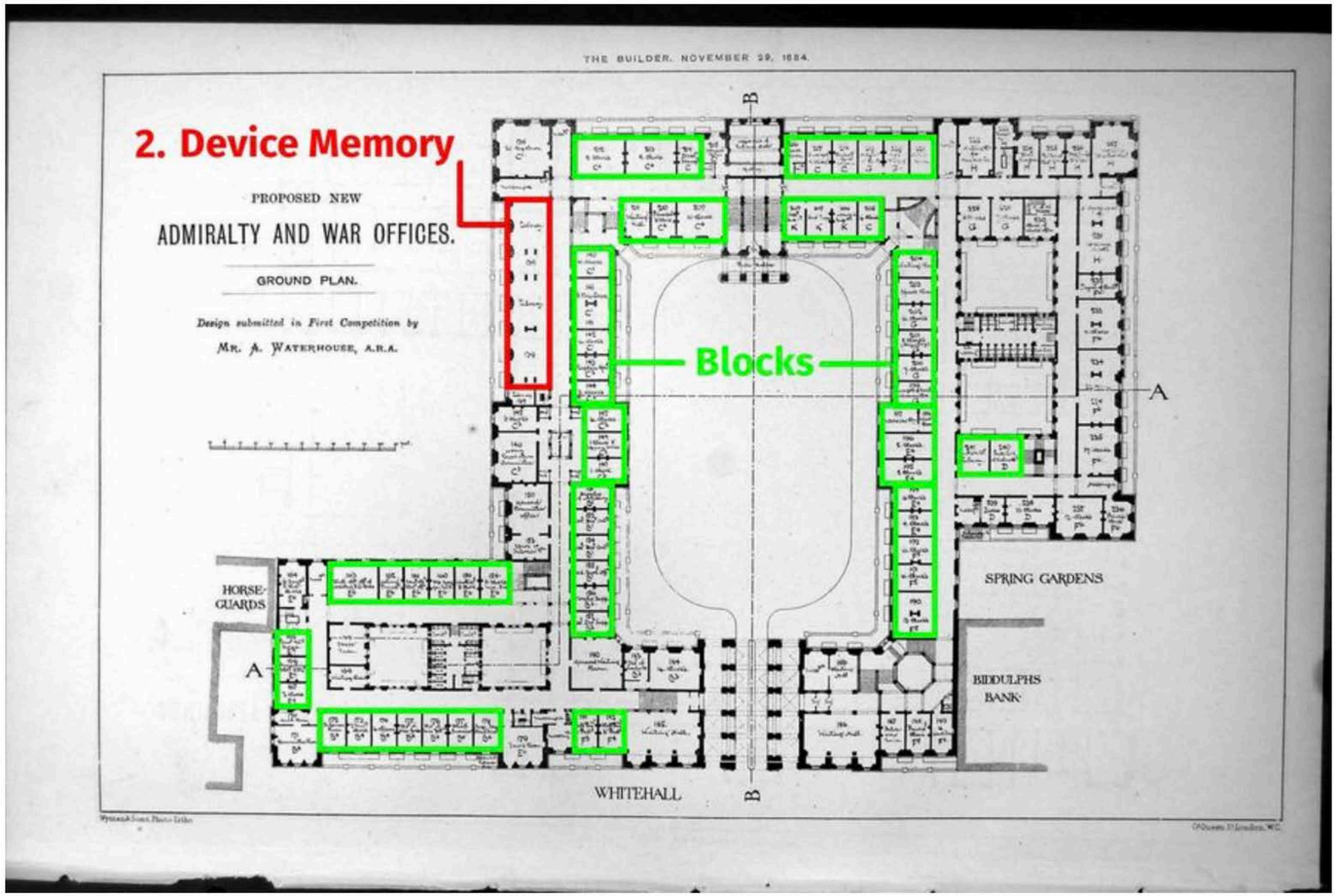
- Main algorithm
- Rasterization pipeline
- GPU
- Summary

What is GPU?

- Hardware that accelerates the graphics pipeline.
- GPU is a cluster of thousands of efficient workers knowing simple arithmetics working in parallel.
- GPU works in “Single Instruction Multiple Data” (SIMD) (as opposed to Single Instruction Single Data (SISD) like CPU)

What is GPU?

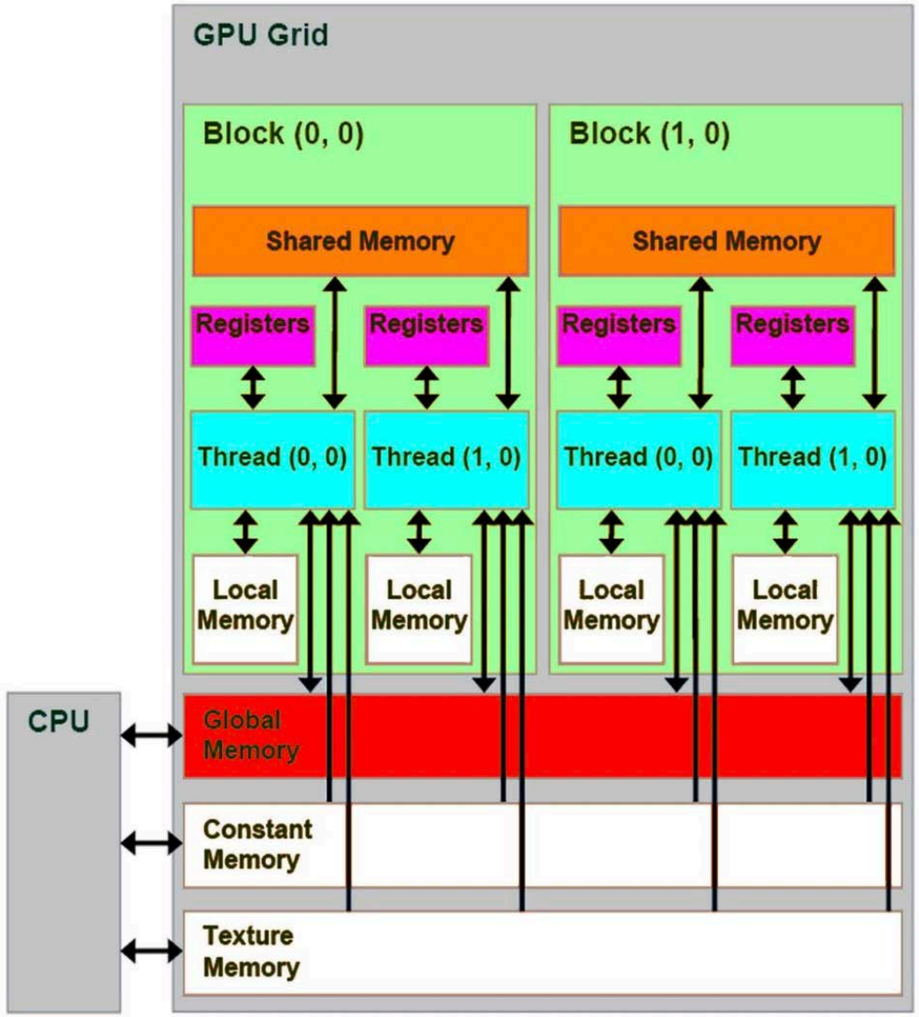


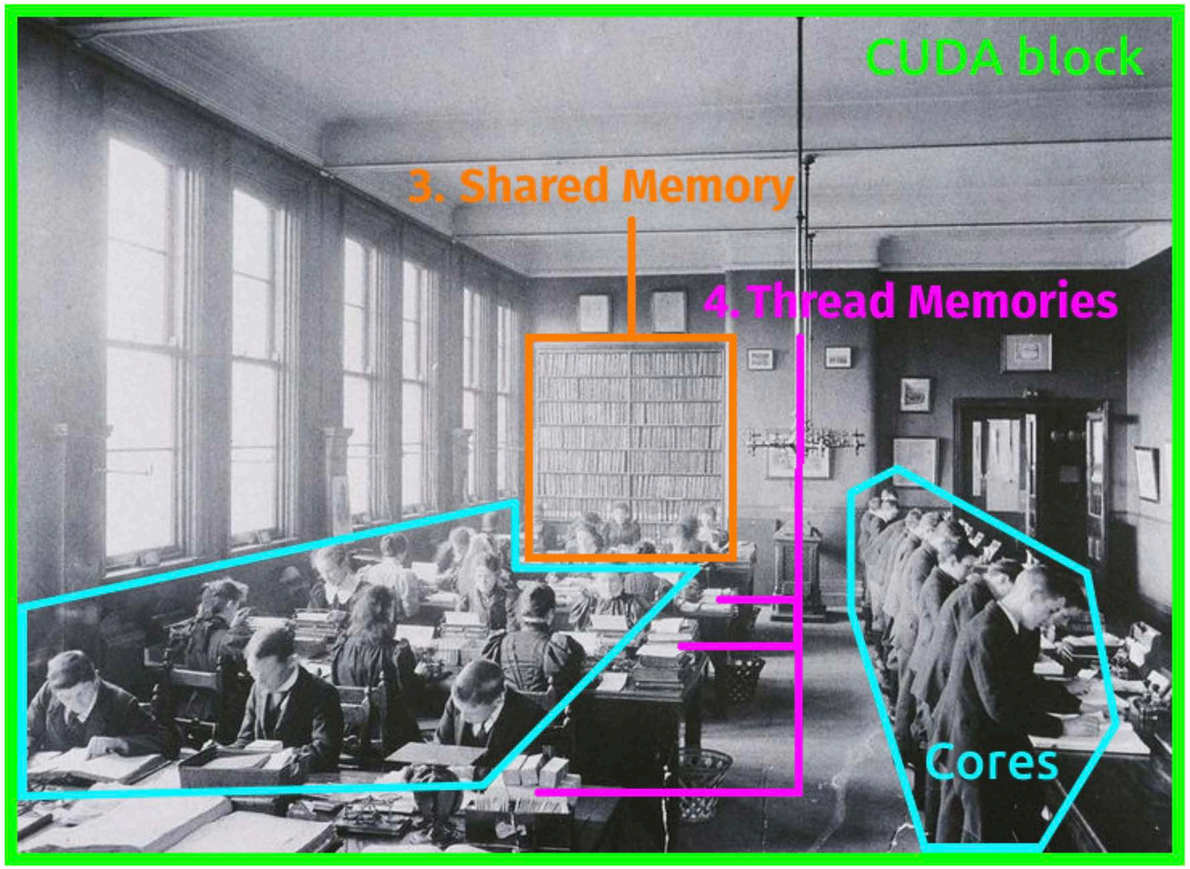


A 19th century office building

(image courtesy Jean Feydy)

What is GPU?

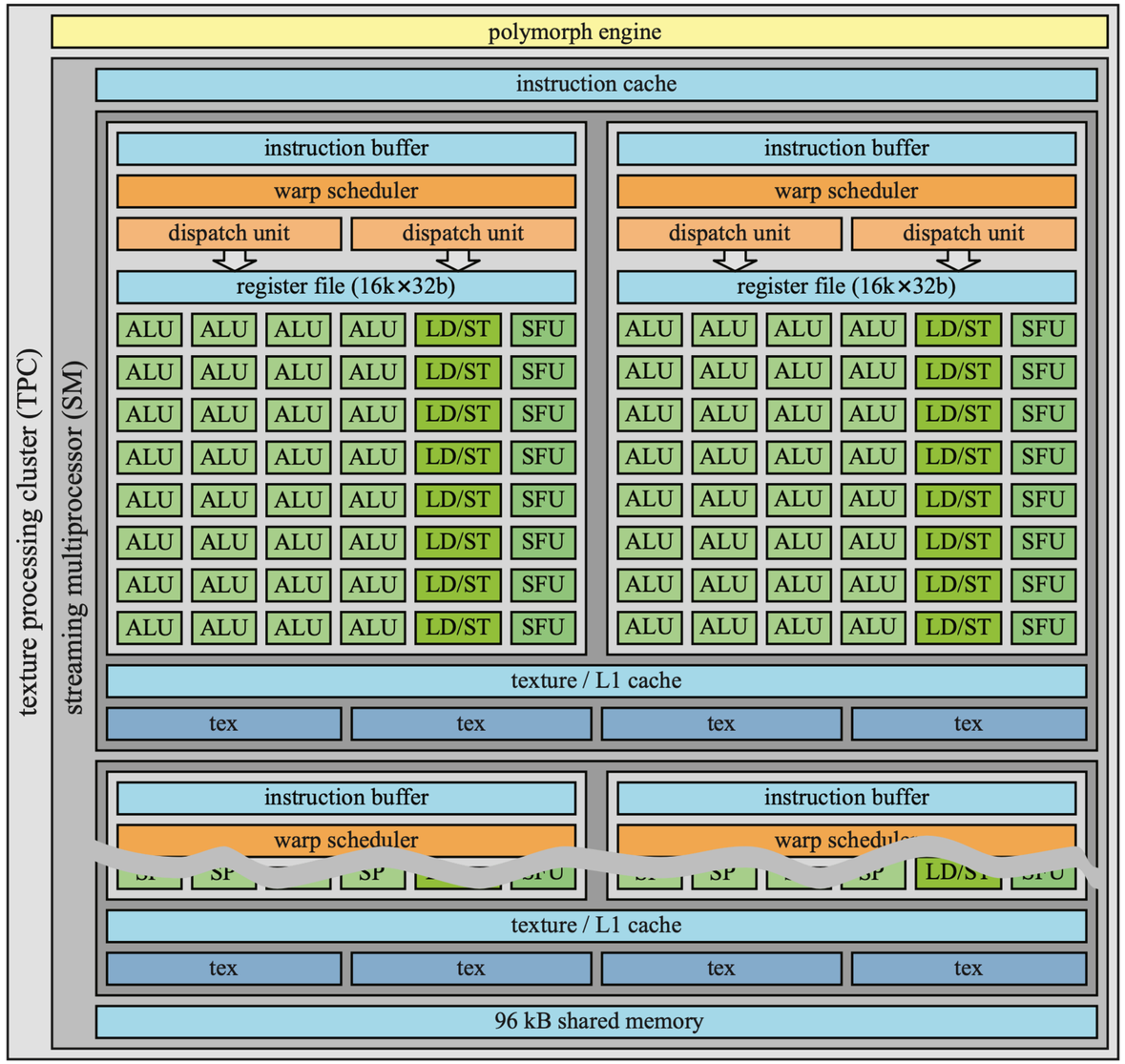


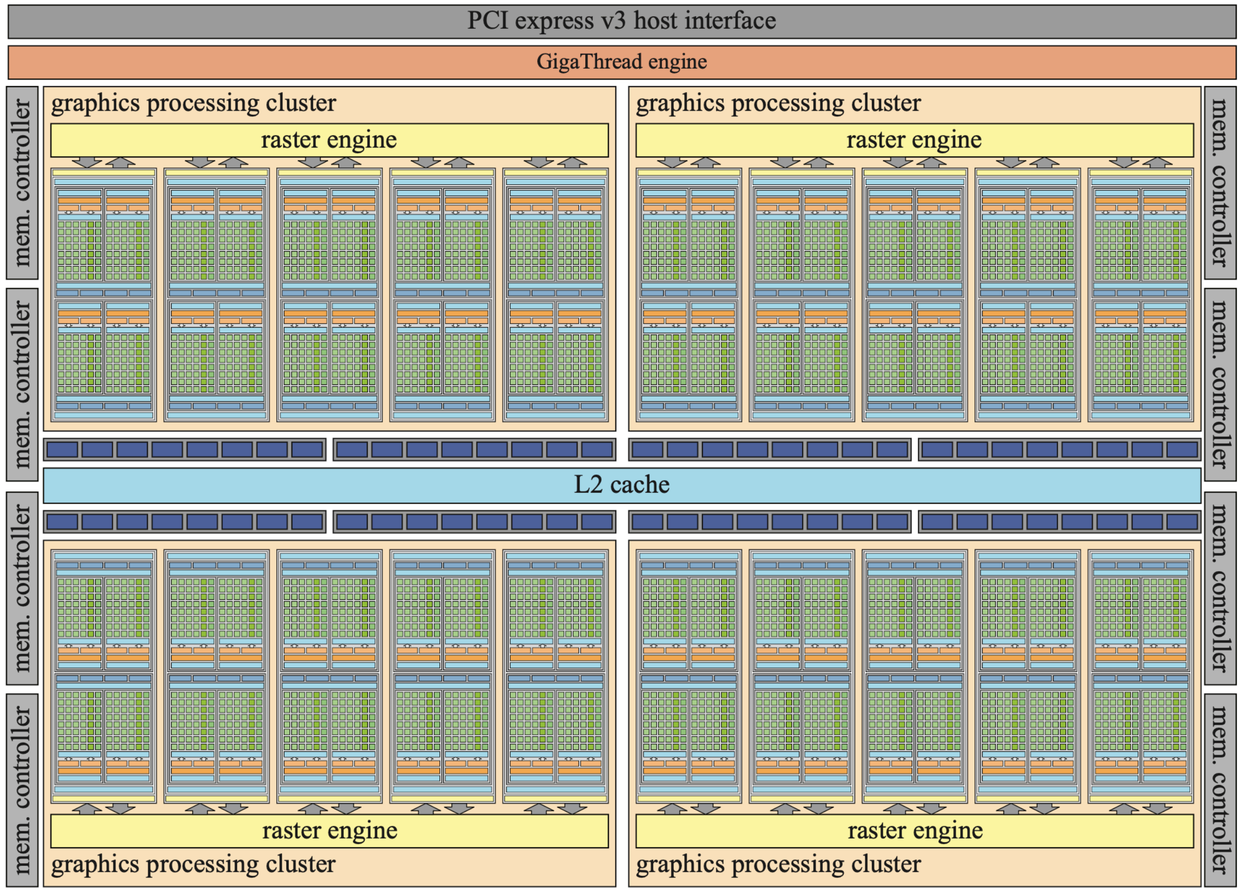


A 19th century office building

(image courtesy Jean Feydy)

What is GPU?





(image from Real-Time Rendering Ch 23)

What is GPU?

- Frequent use of graphics pipeline $\Longrightarrow$ Dedicated hardware
- GPU manufacturers provide application programming interface (API) which is a list of functions allowing us to command GPU
- 90's: GPU are made with fixed functions (e.g. Legacy OpenGL)
- Mid 2000's: Programmable shaders (e.g. Modern OpenGL)

  Definition A ***shader*** is a program that runs on GPU

- Trend: Towards programmability and flexibility
- General purpose GPU (e.g. CUDA), high performance computing, parallel computing, machine learning

Summary

- Main algorithm
- Rasterization pipeline
- GPU
- Summary

Summary

- **Fragments** are triangle-pixel incidence/intersection
- **Rasterization** and **ray tracing** are two ways of looping to find fragments
- A **buffers** is an allocated memory (on the graphics card)
- A **shader** is a program running on GPU
- Describe a shape by a geometry spreadsheet (vertex array object) stored in **vertex buffer** and **index buffer**
- Raster graphics pipeline:

- Implemented in GPU; **OpenGL** is an API to command it.