Lecture 22: Rasterization Pipeline and GPU Hardware

Yuhao Zhu

http://yuhaozhu.com
yzhu@rochester.edu

CSC 259/459, Fall 2024
Computer Imaging & Graphics

The Roadmap

Theoretical Preliminaries

Human Visual Systems

Digital Camera Imaging

Modeling and Rendering

3D Modeling

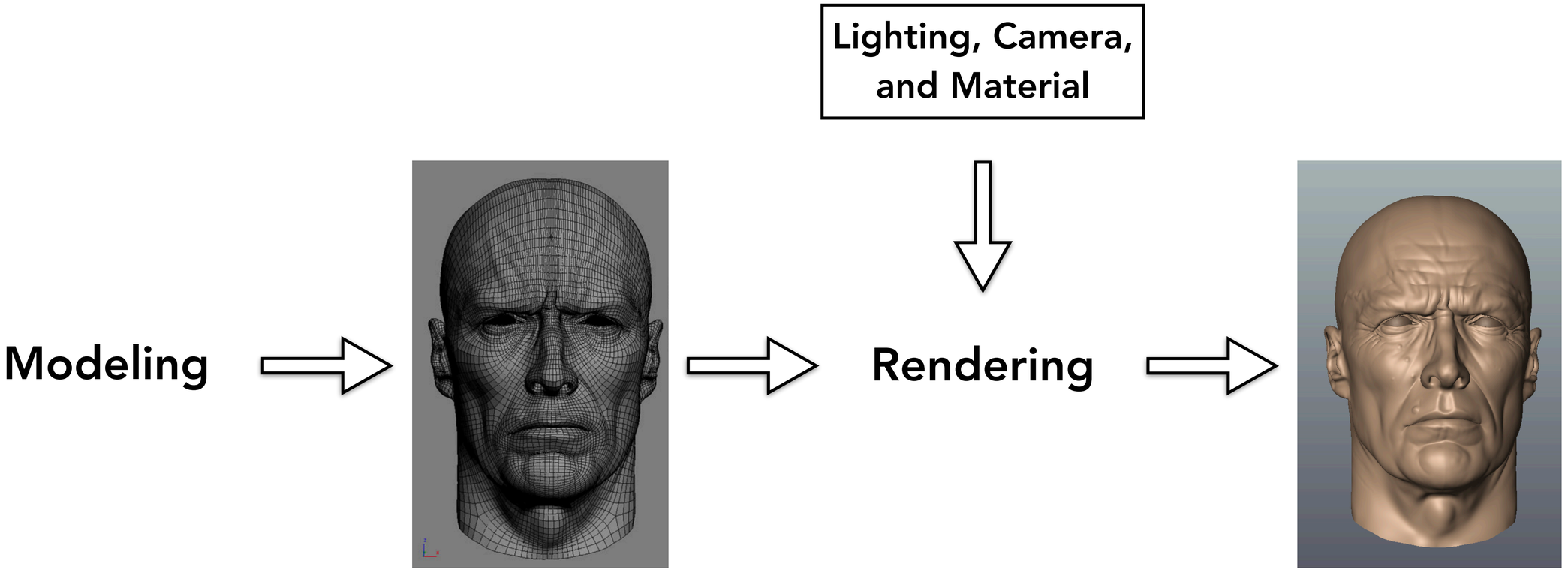
Rasterization and GPU

Ray Tracing

Shading

Rendering in AR/VR

Graphics



https://docs.blender.org/manual/en/dev/render/introduction.html http://www.cgarena.com/freestuff/tutorials/max/thomas_highway/sergeant.html

Rendering Algorithm

Two fundamental problems: visibility and shading

Visibility: what part of the scene is visible by the camera?

- For each image pixel, which point in the scene corresponds to it?
- How many scene points for a pixel?

Shading: how does the visible part look like?

- What's the color of each image pixel?

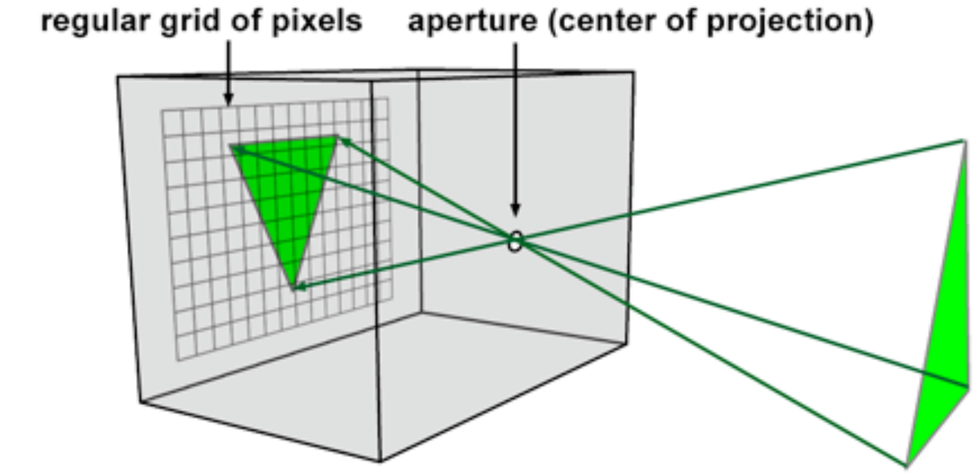
Theoretically shading is independent of visibility, but certain class of visibility algorithms make realistic shading easier/natural to implement.

Visibility Problem

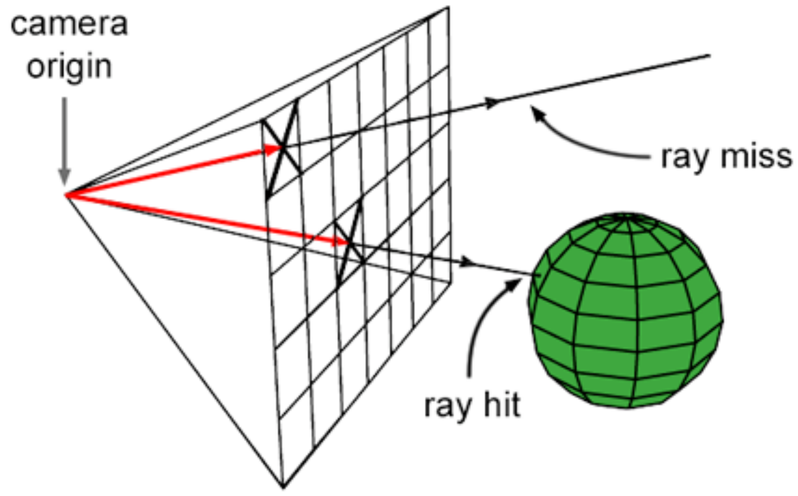
Two fundamental classes of visibility algorithms

- Object-centric (Rasterization)
- Image-centric (Ray tracing)

Given a point P [x, y, z], what's the corresponding pixel coordinates [u, v] on the camera sensor?



Given a pixel [u, v] on the sensor, what's the associated point in the scene?



Visibility Algorithm

Rasterization is generally (much) faster than ray tracing.

Modern GPUs are well-optimized for rasterization, but hardware that supports real-time ray tracing is there (e.g., Nvidia's Turing GPUs).

Ray tracing allows for a natural implementation of realistic shading.

RenderMan (REYES) from Pixar is based on rasterization.

- Considered to be one of the best rasterization algorithm ever to be built
- Today's rasterization pipeline has many similarities with REYES

Pixar now uses RIS, which is purely based on ray tracing.

Shading

Heavily researched; always a speed-vs-realism trade-off.

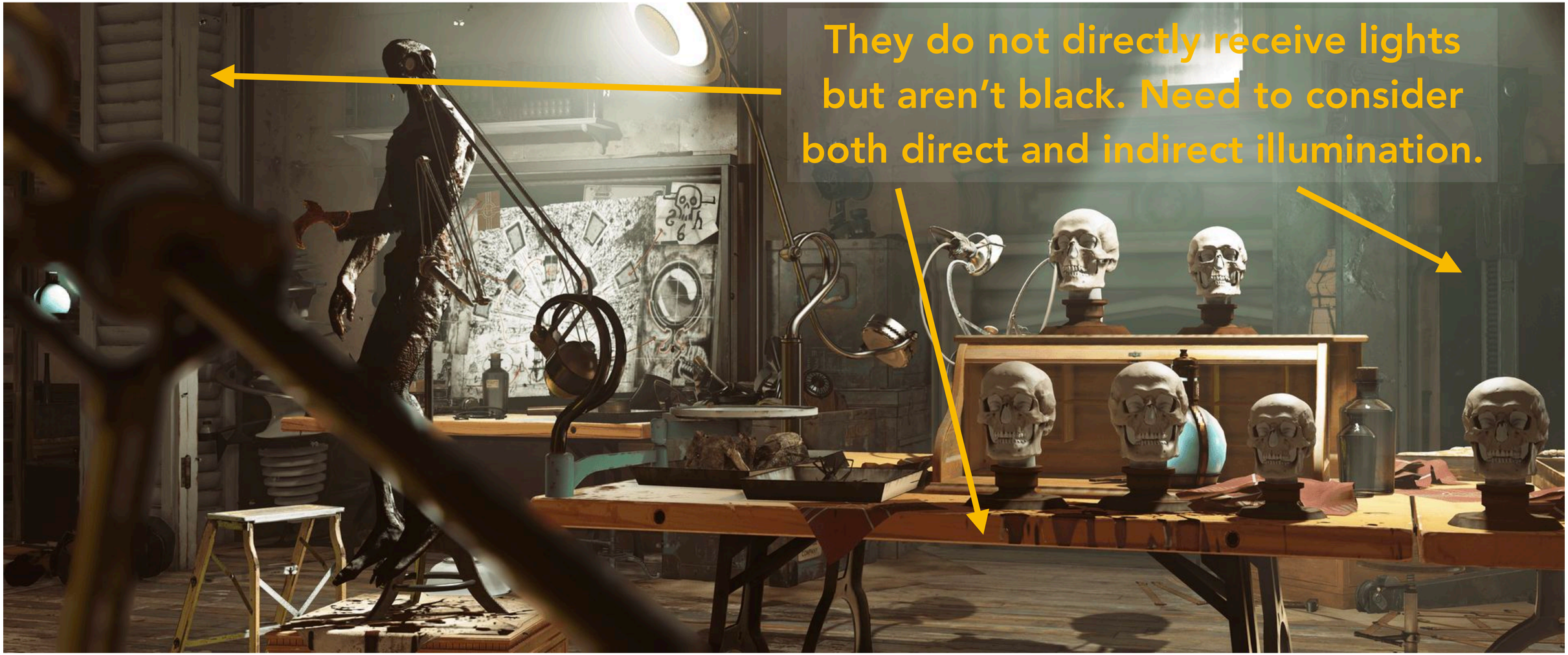
Empirical modeling vs. physics simulation

- Simple solutions, e.g., assigning color to each scene point/triangle + interpolation
- Slightly better: empirical modeling (e.g., Phong model)
- Ultimately, we must simulate physics (e.g., light matter interaction, spectral information)

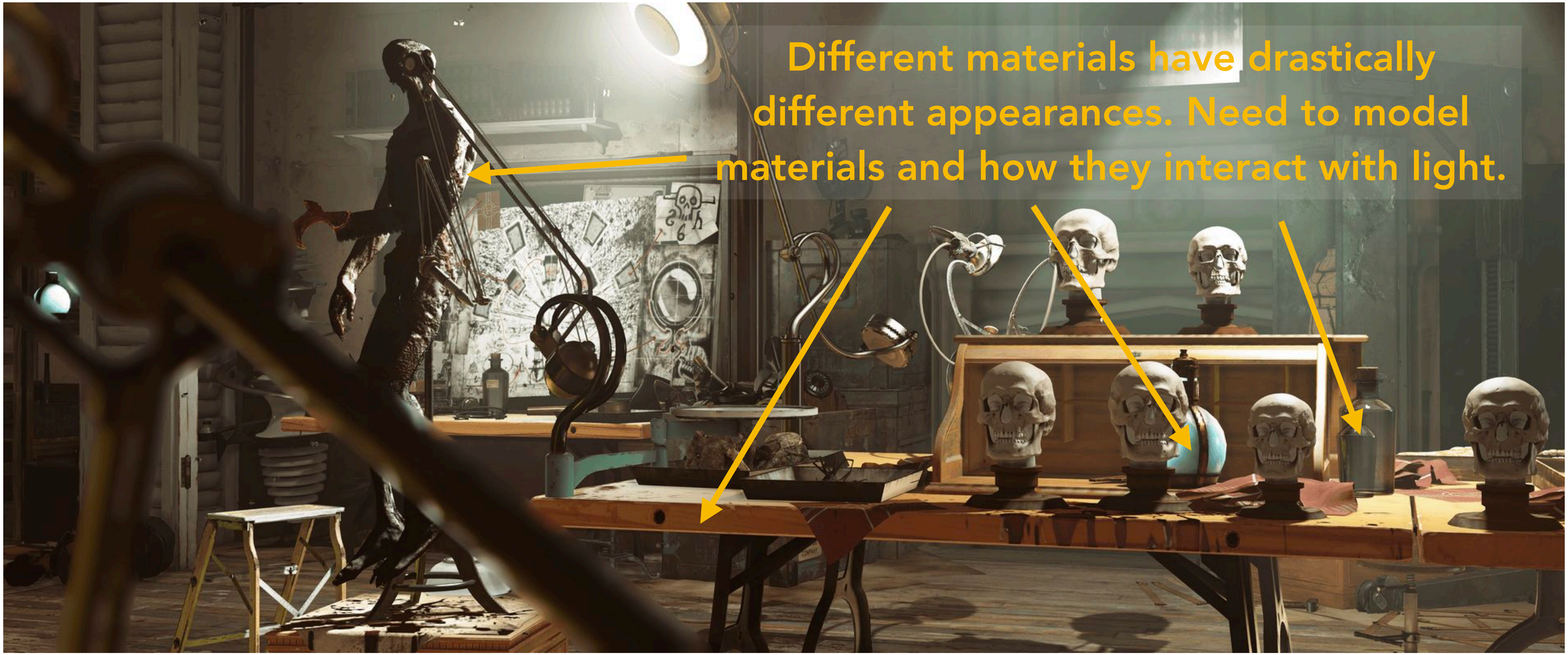
Local vs. global illumination

- Do we consider only direct lighting or also account for indirect illumination (e.g. reflection from other objects), a.k.a., global illumination?

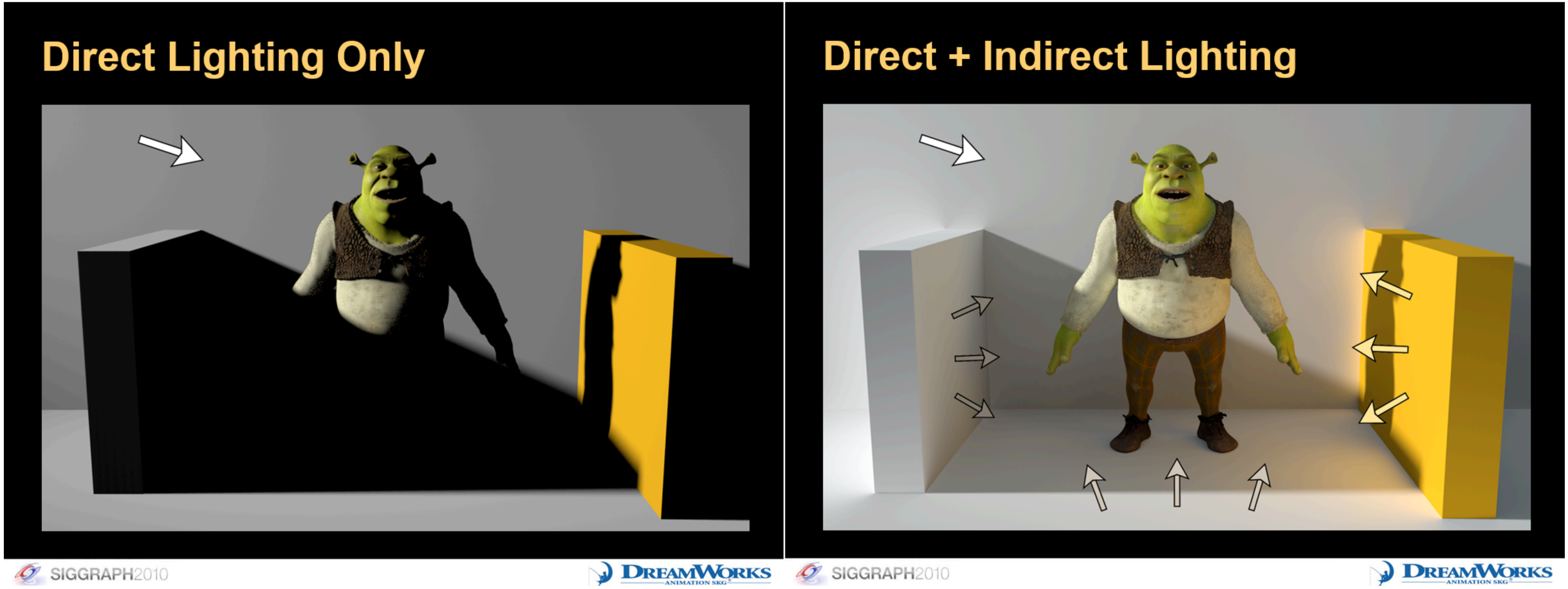
Shading Complexity: Global Illumination



Shading Complexity: Modeling Light-Matter Interaction



Local vs. Global Illumination



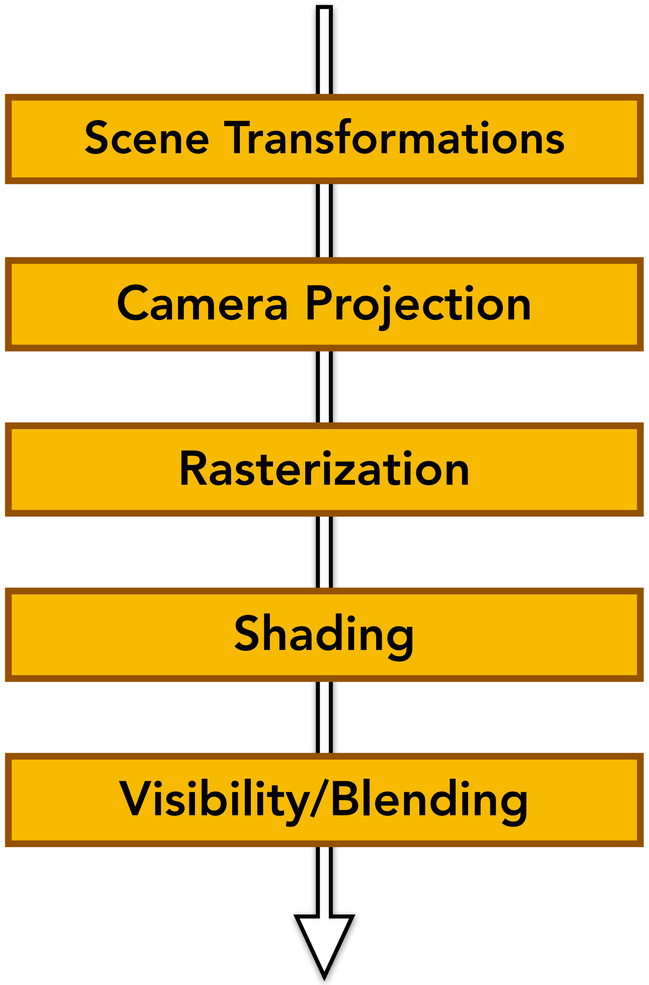
https://colinbarrebrisebois.com/2015/11/06/finding-next-gen-part-i-the-need-for-robust-and-fast-global-illumination-in-games/

https://www.tomlooman.com/lighting-with-unreal-engine-jerome/

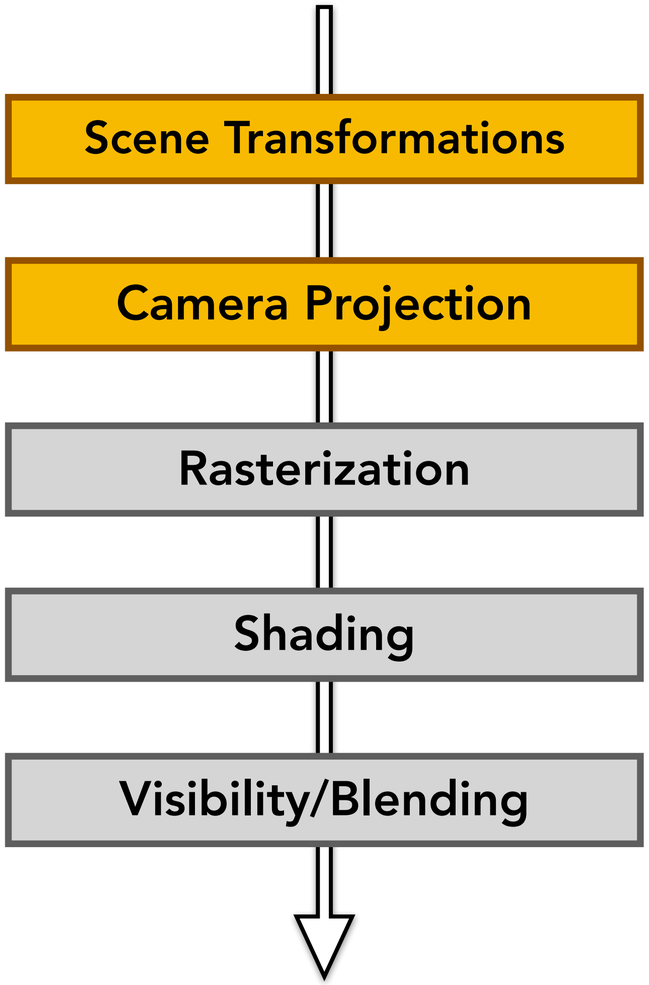
https://www.tomlooman.com/lighting-with-unreal-engine-jerome/

Rasterization Pipeline

Rasterization-based Rendering



Rasterization-based Rendering



Converting scene objects to camera screen space

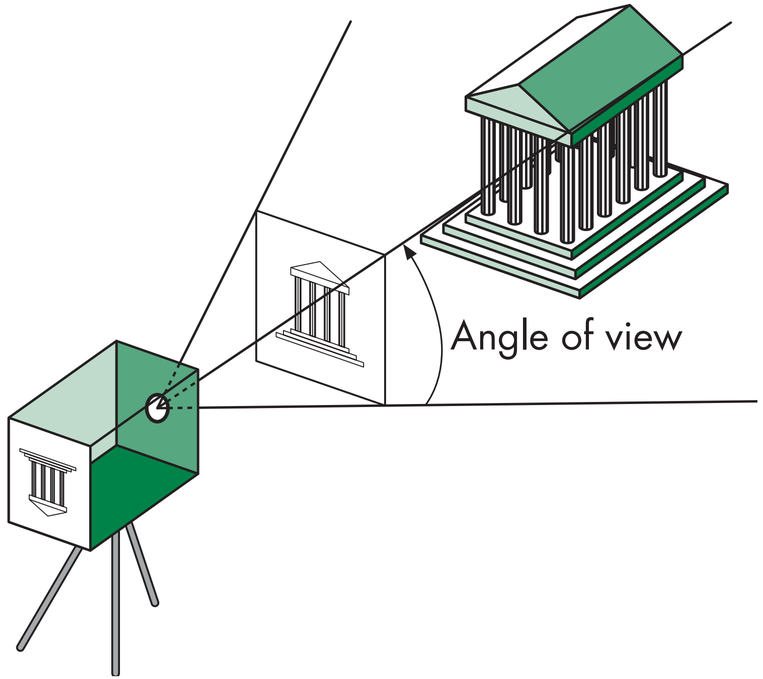
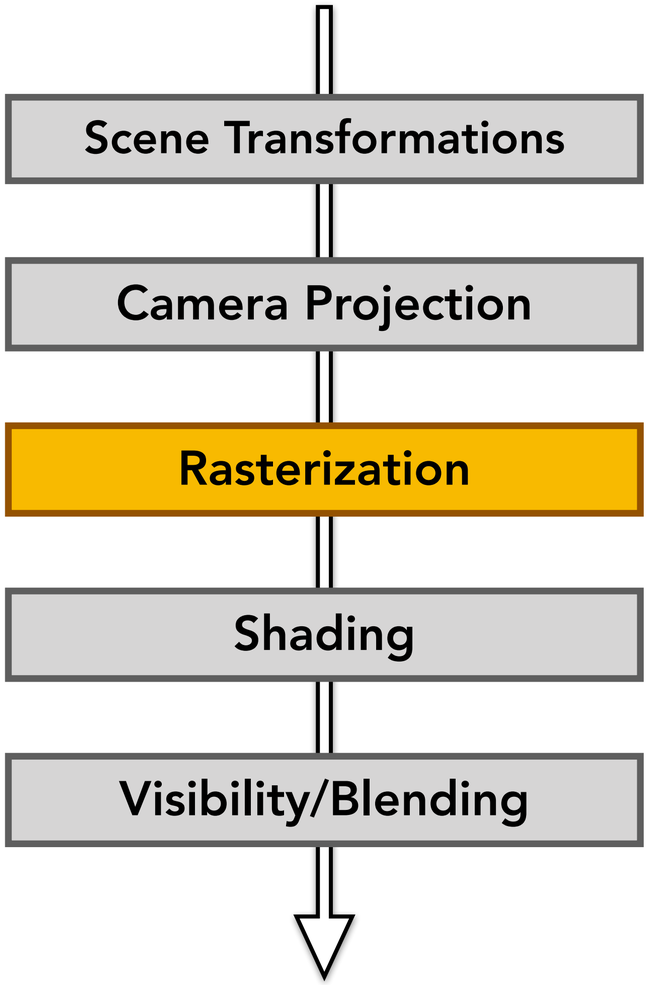
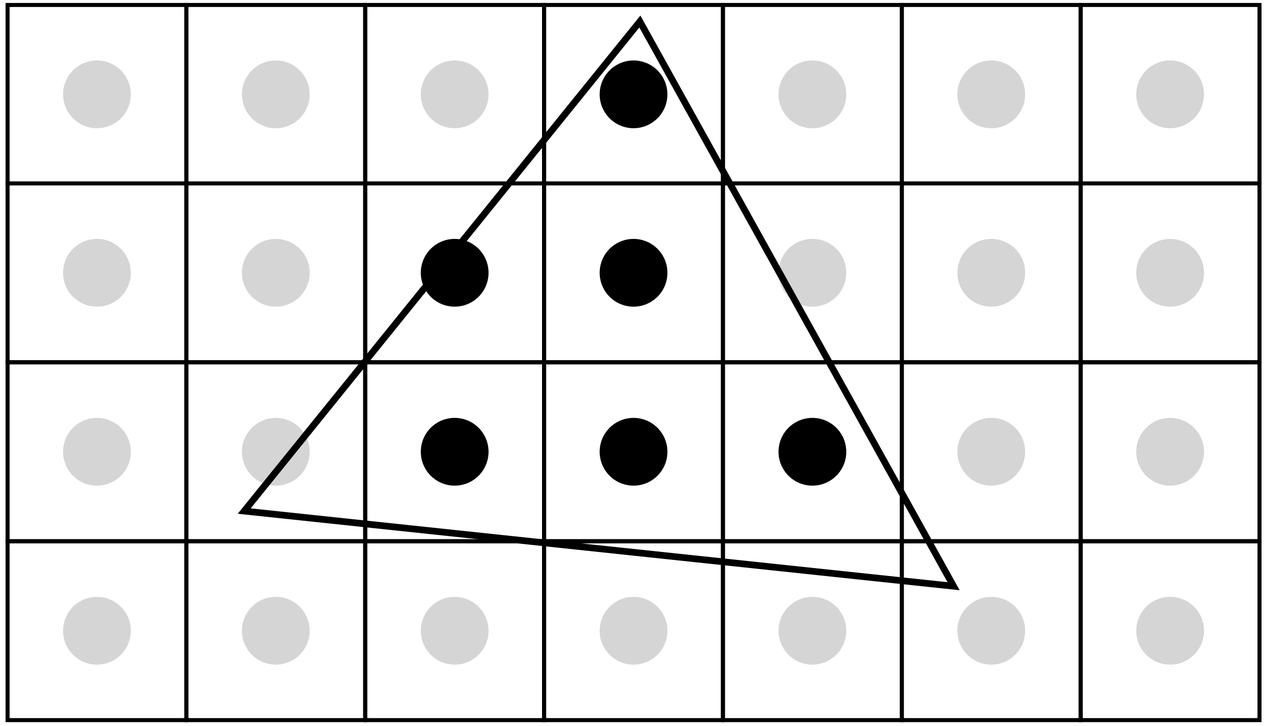


FIGURE 5.34 Specification of a view volume.

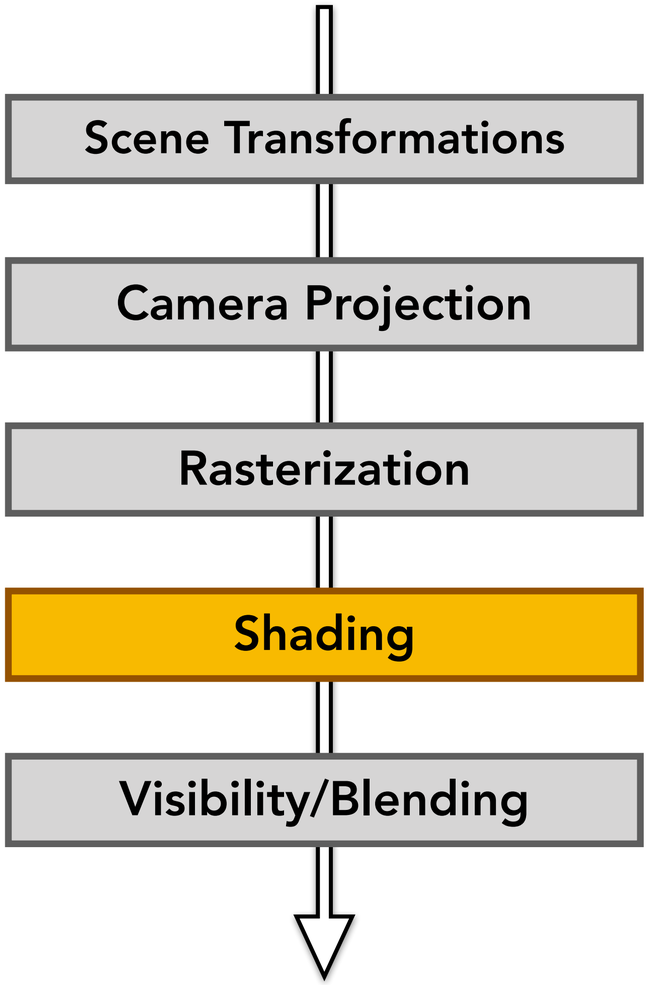
Rasterization-based Rendering



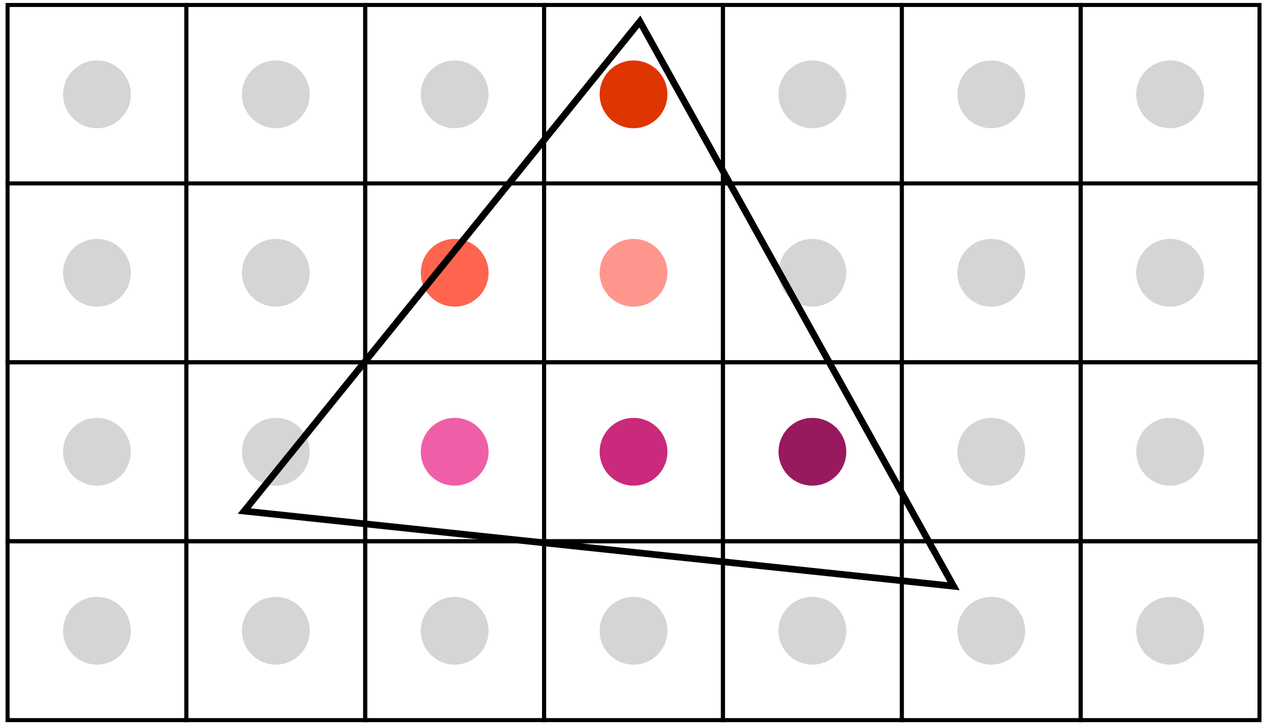
Which pixels are covered by each triangle?



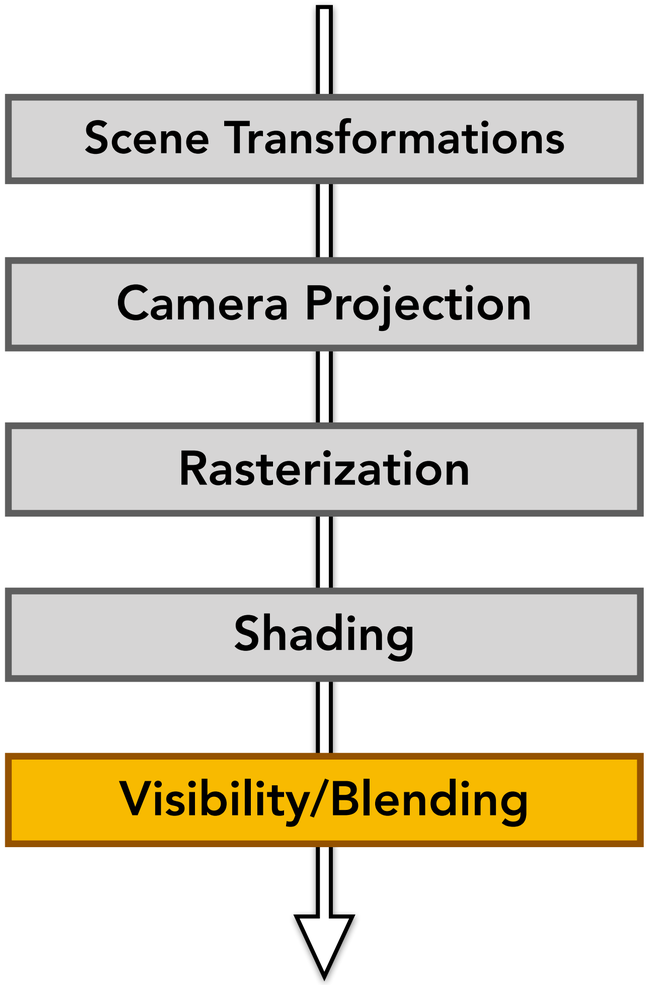
Rasterization-based Rendering



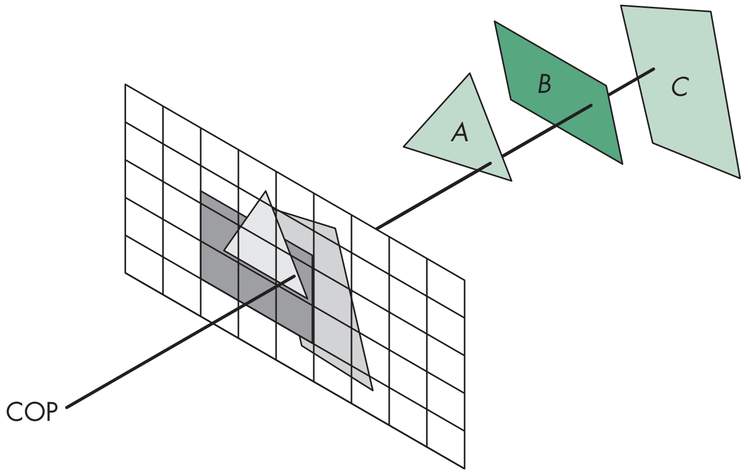
What's the color of each pixel?

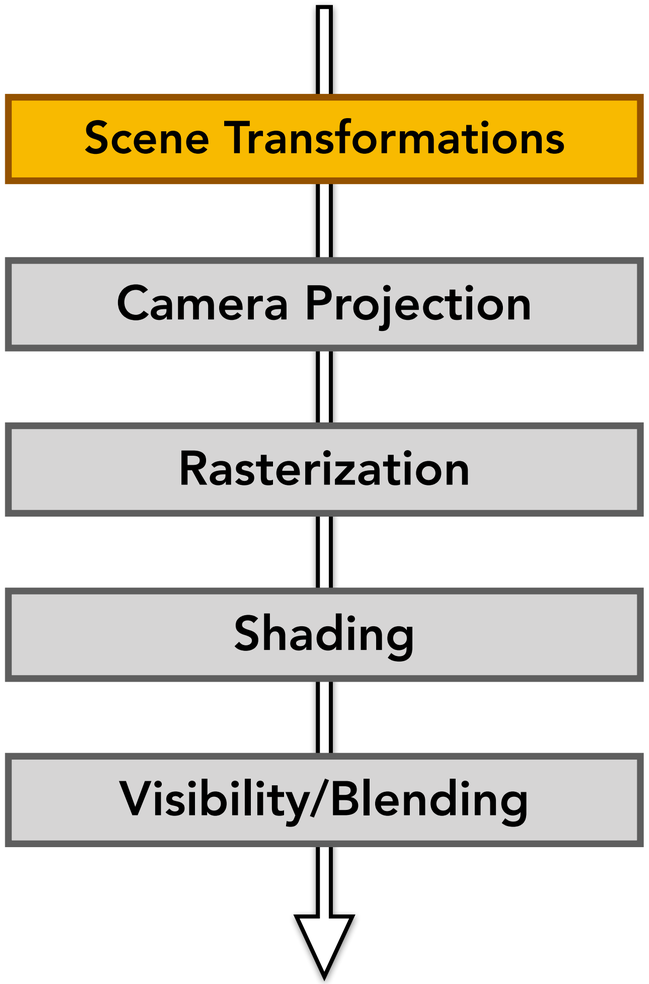


Rasterization-based Rendering



How to deal with multiple scene points mapped to the same pixel?





Scene Transformation

Scene Transformation

For convenience and for reusing the same objects across scenes.

The two killeroos are exactly the same object, but are placed differently in the same scene.

Define the mesh of the killeroo once with respect to its local coordinate system, and transform it properly when place it in the world coordinate system.

Example

A scene description file from **pbrt**,
a pedagogical rendering engine.

Different transformations (translations)
when placed in the scene

```
Translate 100 200 -140
    Include "geometry/killeroo.pbrt"
    Material "plastic" "color Ks" [.3 .3 .3]
        "float roughness" [.15]
Translate -200 0 0
    Include "geometry/killeroo.pbrt"
```

Object description in its local
coordinate system (not shown here)

What Local Coordination Systems Are There?

Scene

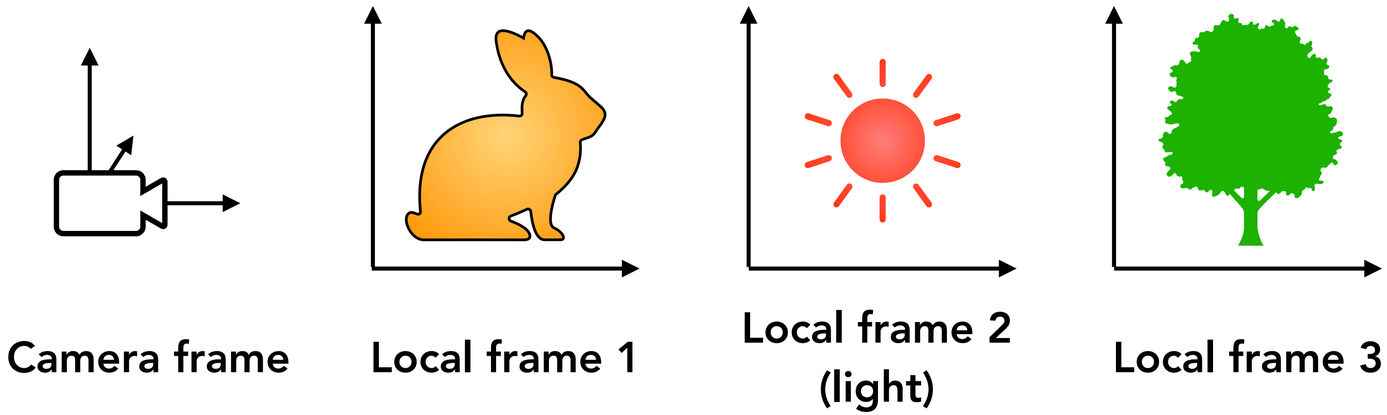
Objects

Light sources

- Point light (shapeless)
- Area light
- Distant light
- Arbitrary shapes

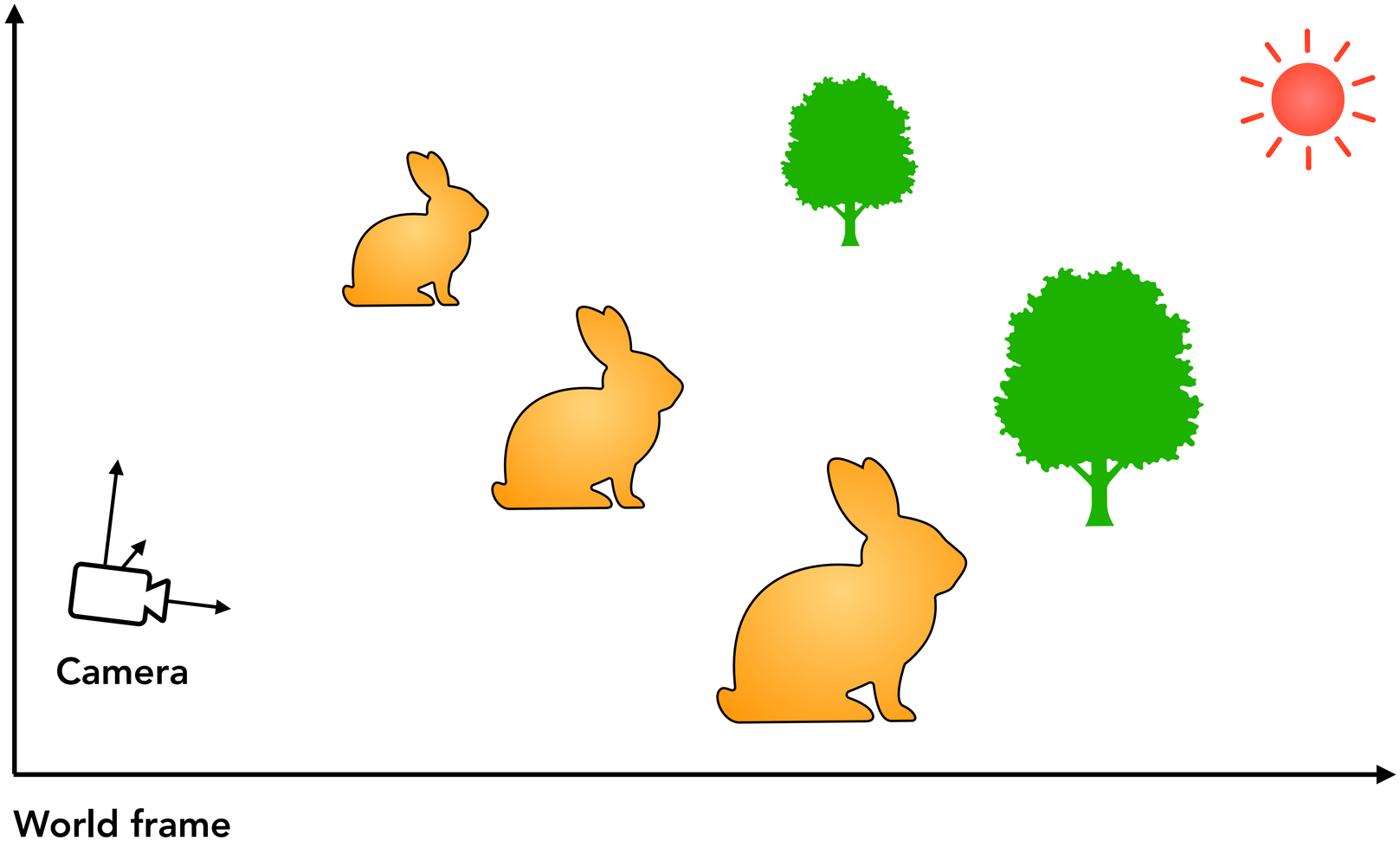
Camera

- A special local frame, where everything else eventually has to be translated to.



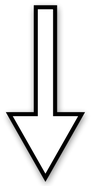
Scene Transformations

Local to world transformations (3D to 3D)

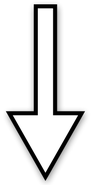


Scene Transformations

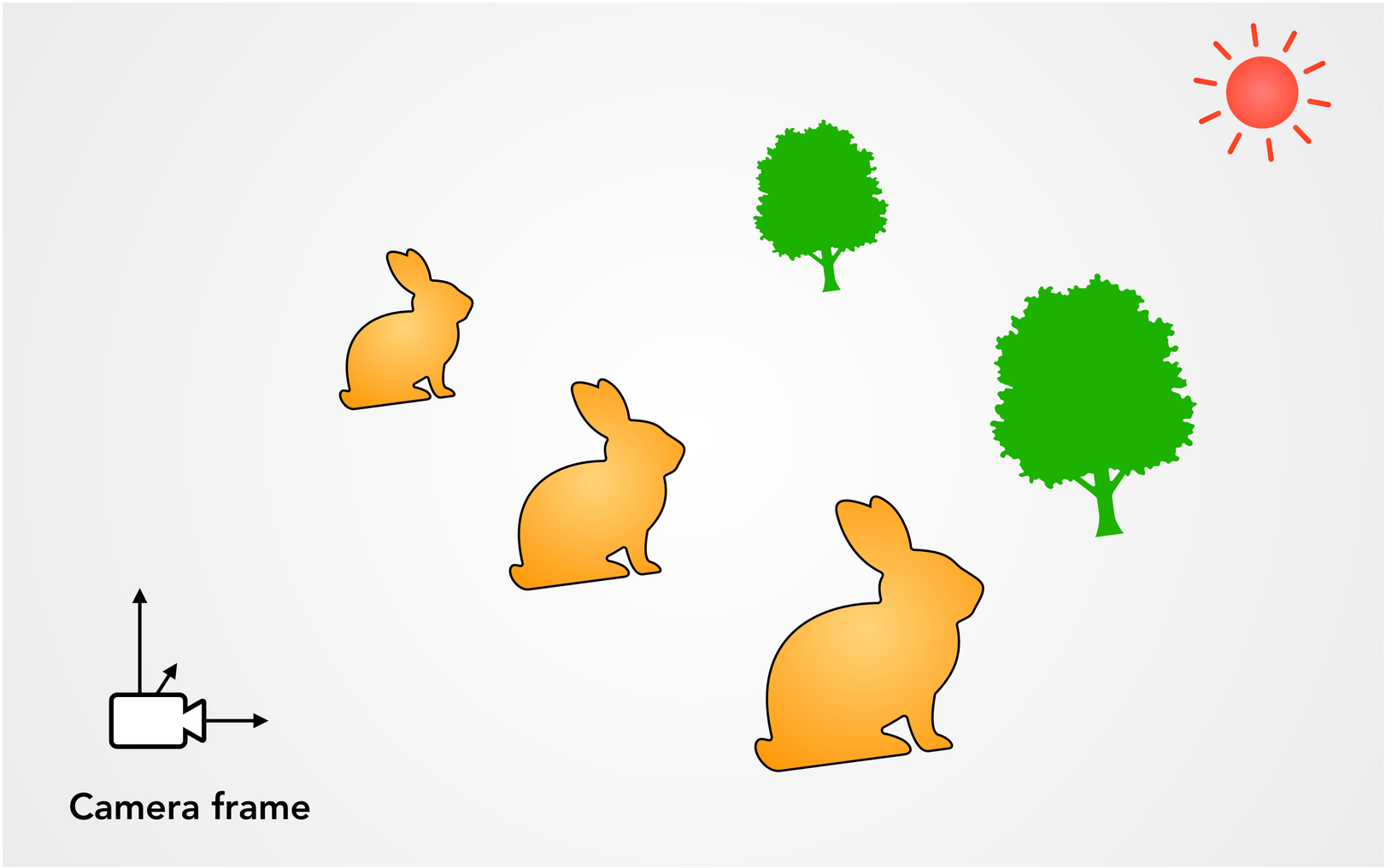
Local to world transformations (3D to 3D)

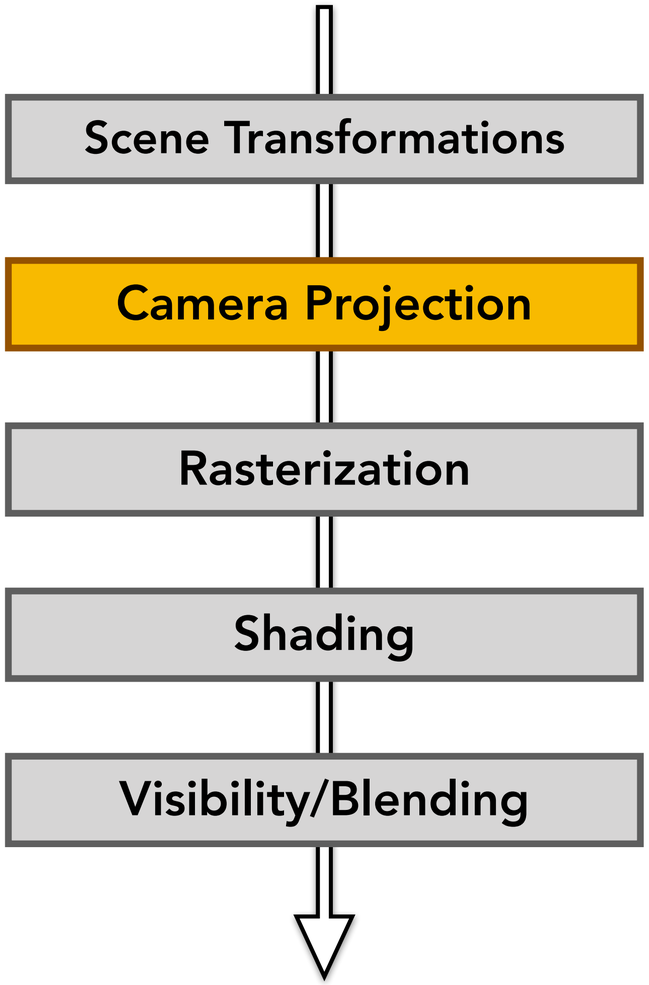


World to camera transformation (3D to 3D)



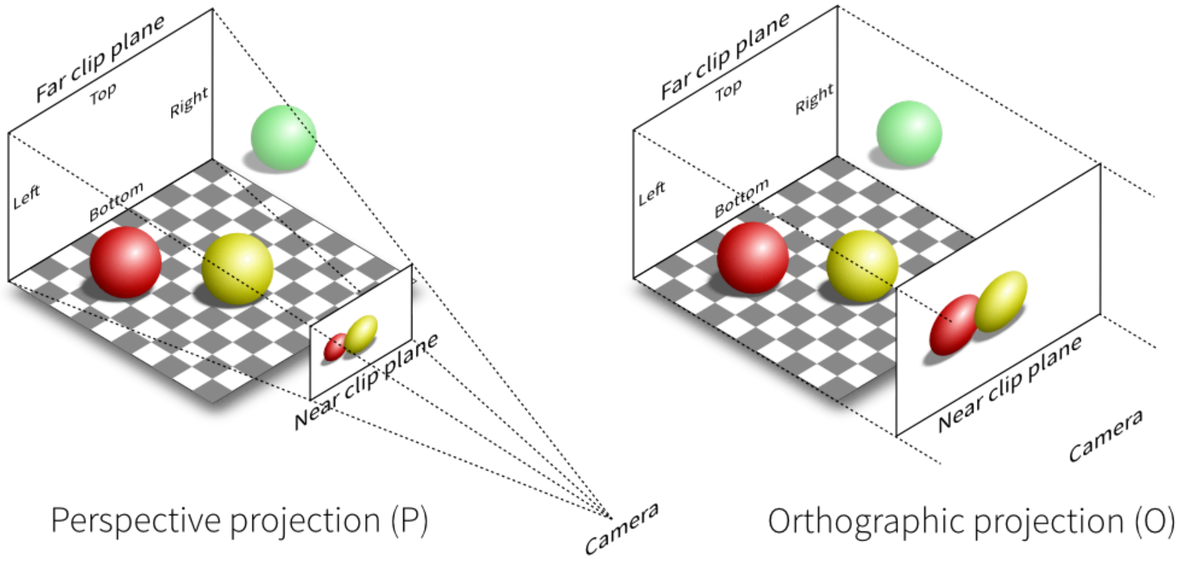
Camera Projection (3D to 2D)

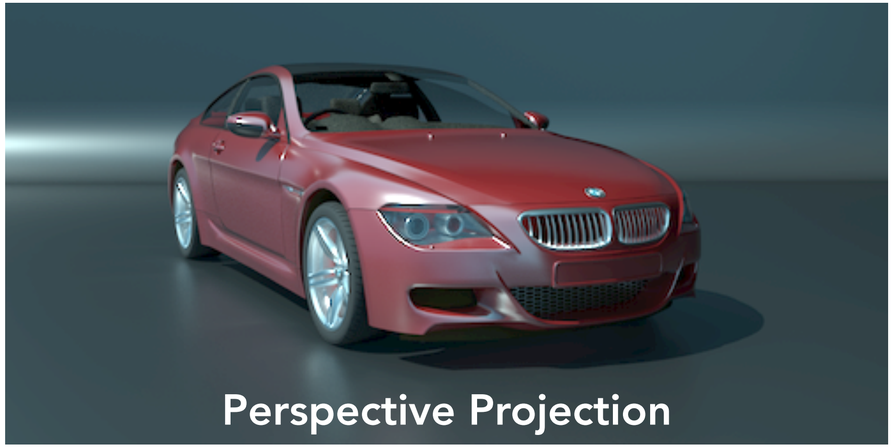


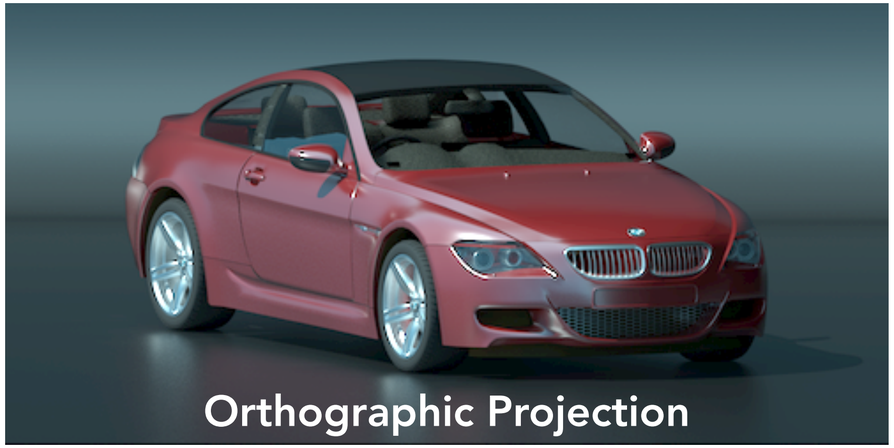


Camera Projection

Camera Projections: Where 3D Becomes 2D



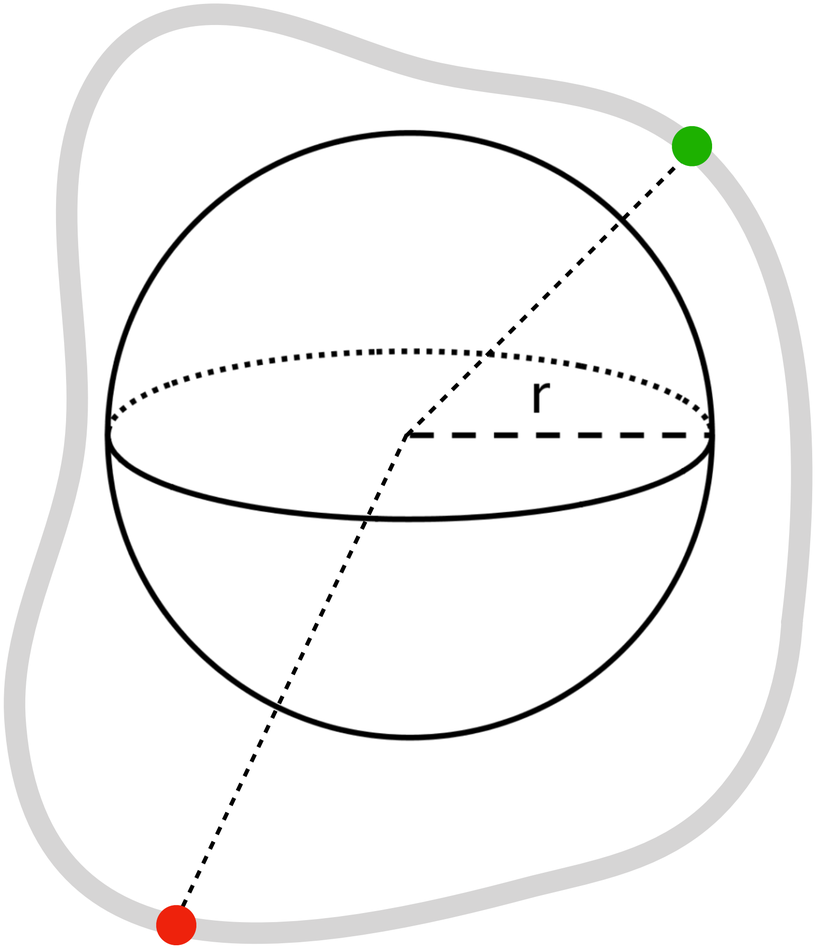




http://www.pbr-book.org/3ed-2018/Camera_Models/Environment_Camera.html


Quests
Menu (F10)
Allies (F11)
Chat (F12)
265
170
7/20
No Upkeep
+10
+10
Barracks
Armor:
5
1200 / 1200

Environmental Camera Projection



Environmental camera

http://www.pbr-book.org/3ed-2018/Camera_Models/Environment_Camera.html

Camera Projection: Where 3D Becomes 2D

Fundamental question: given a point P [x, y, z], what's the corresponding pixel coordinates [u, v], if any, on the camera sensor?

- A point might not been seen by the sensor because of occlusion and/or FOV.

There are many ways to project a 3D point to a 2D pixel. The most common one is "**perspective projection**".

- It simulates a pinhole camera model, which is roughly how human eyes work; many cameras are built to mimic human eyes.
- But there are other projections that you can implement (after all, graphics is just simulation), and many cameras that are built not to mimic human eyes (e.g., fish-eye cameras).

Convention: Placing Image Plane Before Camera

We assume the sensor is in front of the pinhole — not possible physically, but simplifies drawing.

- Of course the image is not upside down anymore.
- Scene points could be either before or after the image plane, i.e, does not artificially restrict where a scene point can be.

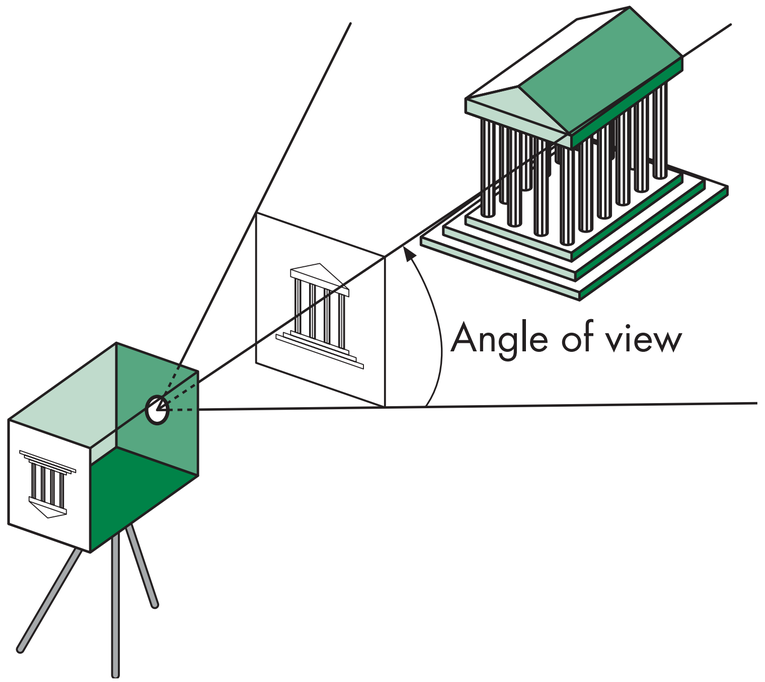


FIGURE 5.34 Specification of a view volume.

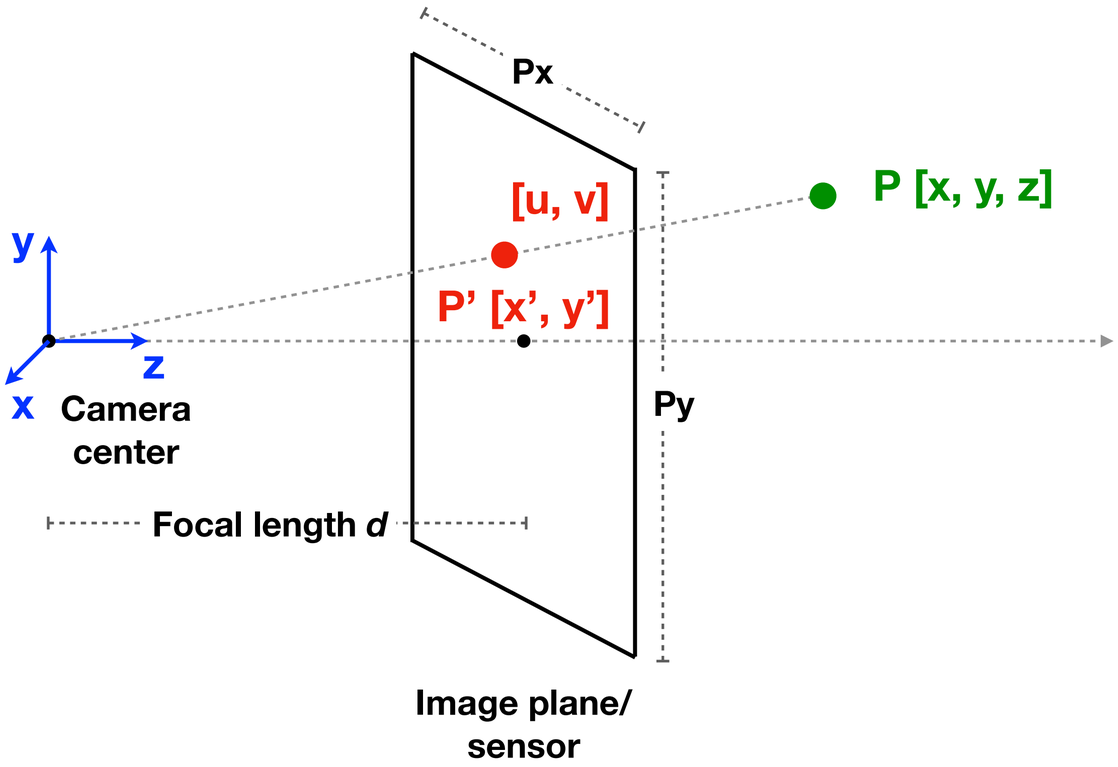
Perspective Projection

Goal: convert P [x, y, z] to pixel coordinates [u, v] on the sensor (with H x W pixels and a focal length *d)* using a **transformation matrix**.

We will do that in two general steps (many caveats will be discussed later):

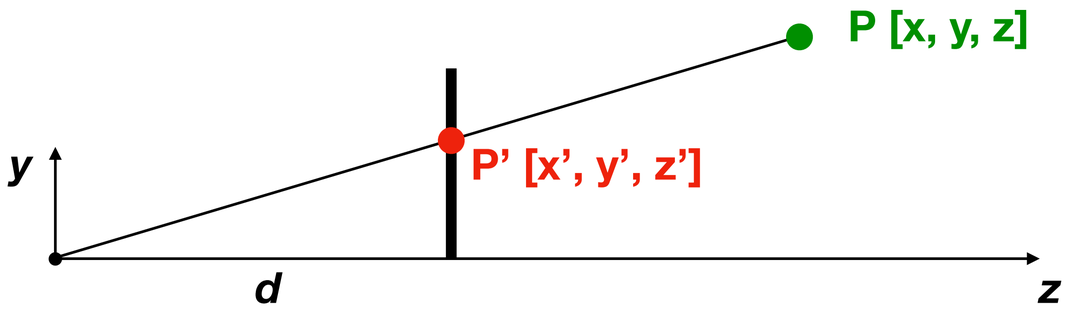
- Perspectively project P[x, y, z] to P'[x', y', d] in the image plane (still in the camera space).
- Convert P' to the pixel coordinates [u, v].

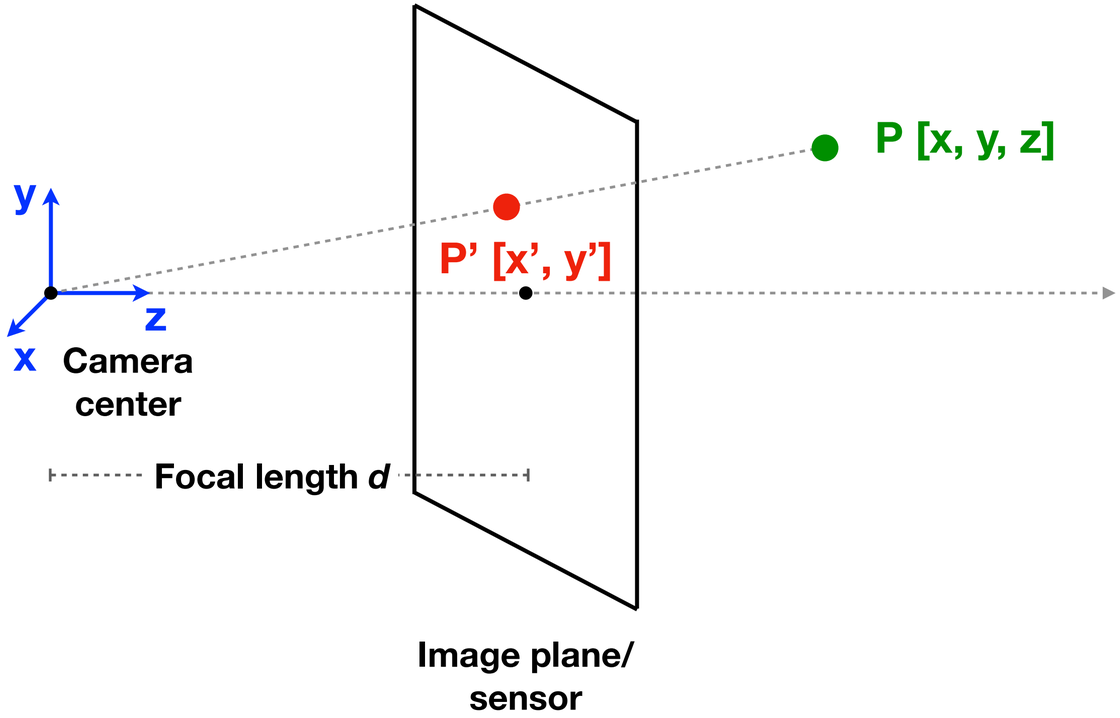
Convention: camera looks down z and looks up to y. Positive z is the viewing direction.



Perspective Projection

$$x' = \frac{x}{z}d \quad y' = \frac{y}{z}d \quad z' = d$$





Perspective Projection Matrix

$$x' = \frac{x}{z}d \quad y' = \frac{y}{z}d \quad z' = d$$

$$[x, y, z, 1] \; x \begin{bmatrix} T_{00}, T_{01}, T_{02}, T_{03} \\ T_{10}, T_{11}, T_{12}, T_{13} \\ T_{20}, T_{21}, T_{22}, T_{23} \\ T_{30}, T_{31}, T_{32}, T_{33} \end{bmatrix} = [x', y', z', 1]$$

$$x' = xT_{00} + yT_{10} + zT_{20} + T_{30} = xd/z$$

T_{00}: d/z, T_{10}: 0, T_{20}: 0, T_{30}: 0

No T_{00}, T_{10}, T_{20}, T_{30} would satisfy this **universally**!

Perspective Projection Matrix

$$x' = \frac{x}{z}d \quad y' = \frac{y}{z}d \quad z' = d$$

$$[x, y, z, 1] \; x \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [x'k, y'k, z'k, k]$$

$$= [xd, yd, zd, z] \Rightarrow [\, xd/z, yd/z, d \,]$$

Homogeneous coordinates Cartesian coordinates

$$k = xT_{03} + yT_{13} + zT_{23} + T_{33} = z$$

0 0 1 0

$$y'k = yd = xT_{01} + yT_{11} + zT_{21} + T_{31}$$

0 d 0 0

$$x'k = xd = xT_{00} + yT_{10} + zT_{20} + T_{30}$$

d 0 0 0

$$z'k = dk = dz = xT_{02} + yT_{12} + zT_{22} + T_{32}$$

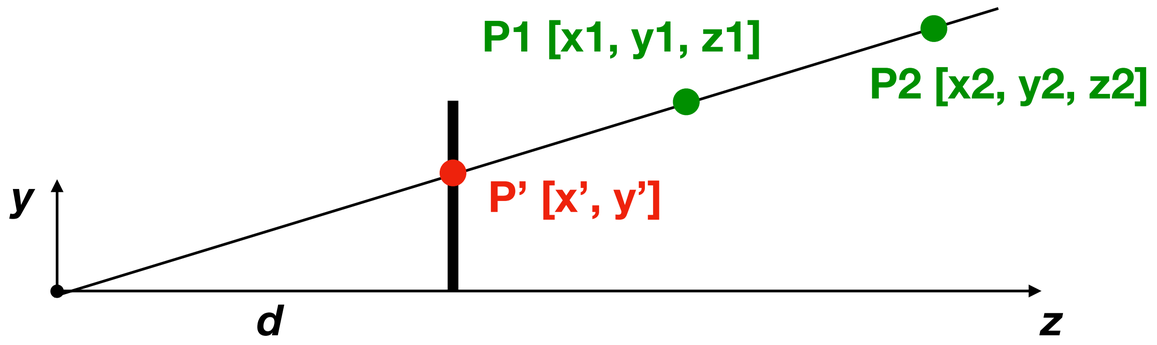
0 0 d 0

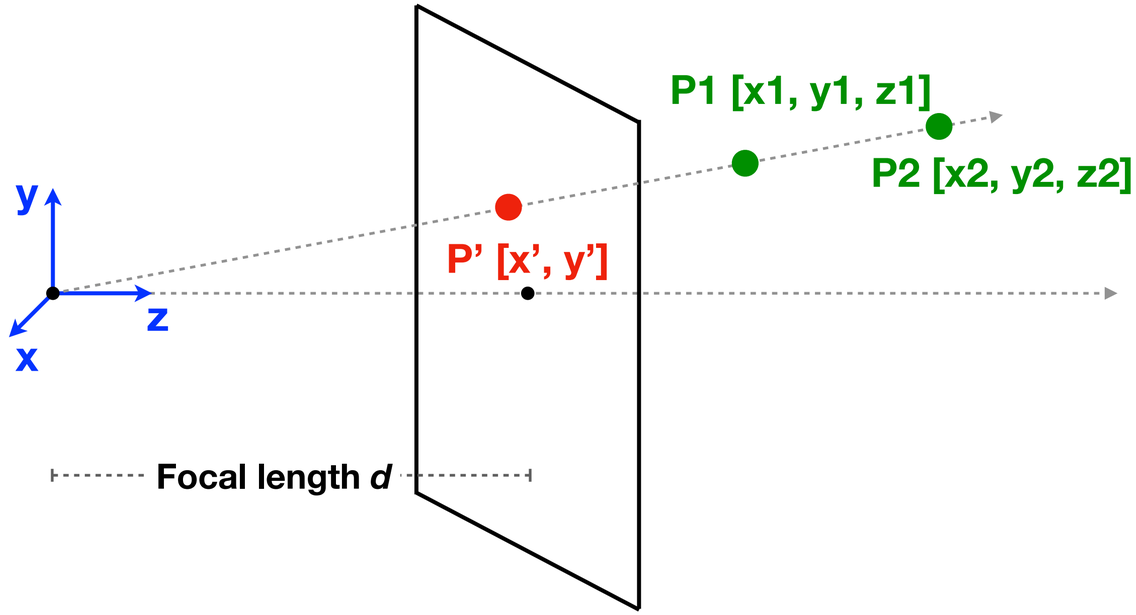
Mind the Z-Axis

Our matrix so far will always translate z-coordinate of any P to the same z′ = d. Good?

P1 and P2 are projected to the same point P′, but P1 is visible and P2 is not: critical for a rendering engine to know.

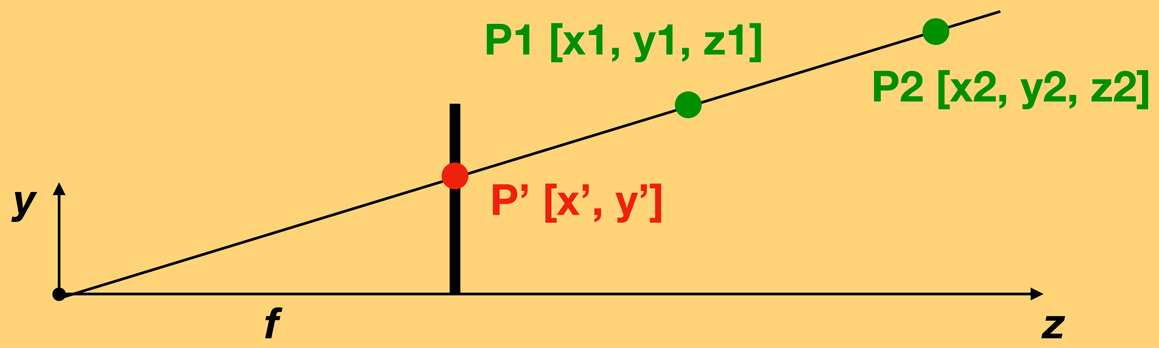
Somehow we need to make sure z1′ < z2′ after projection.





Maintaining Z-Order: Try 1

$$x' = \frac{x}{z} f \quad y' = \frac{y}{z} f \quad z' = z$$



$$[x, y, z, 1] \; x \begin{bmatrix} f & 0 & T_{02} & 0 \\ 0 & f & T_{12} & 0 \\ 0 & 0 & T_{22} & 1 \\ 0 & 0 & T_{32} & 0 \end{bmatrix} = [x'k, y'k, z'k, k]$$

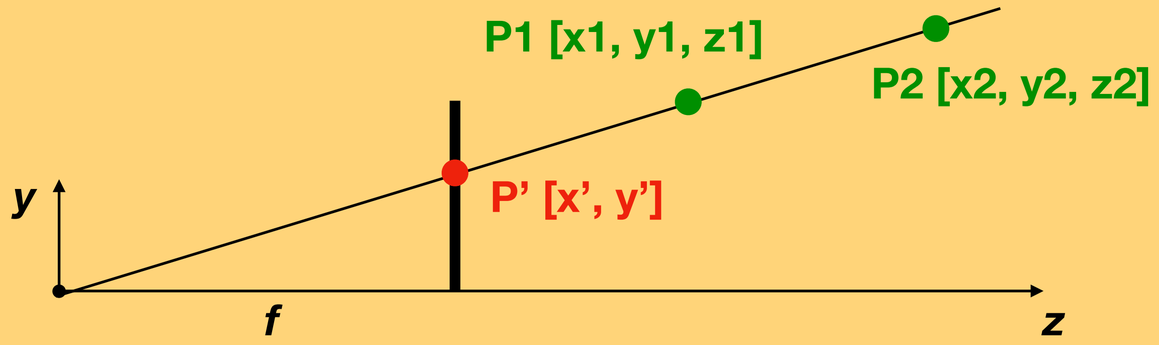
$$z'k = zk = z^2 = xT_{00} + yT_{10} + zT_{20} + T_{30}$$

Try 1: keep z the same before and after transformation

Problem: No one single matrix that universally works for all possible z values

Maintaining Z-Order: Try 2

$$x' = \frac{x}{z} f \quad y' = \frac{y}{z} f \quad z' = Cz$$



$$[x, y, z, 1] \; x \begin{bmatrix} f & 0 & T_{02} & 0 \\ 0 & f & T_{12} & 0 \\ 0 & 0 & T_{22} & 1 \\ 0 & 0 & T_{32} & 0 \end{bmatrix} = [x'k, y'k, z'k, k]$$

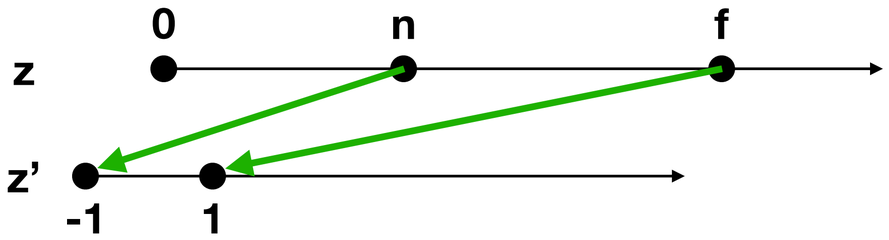
Try 2: scale z with a constant, say C.

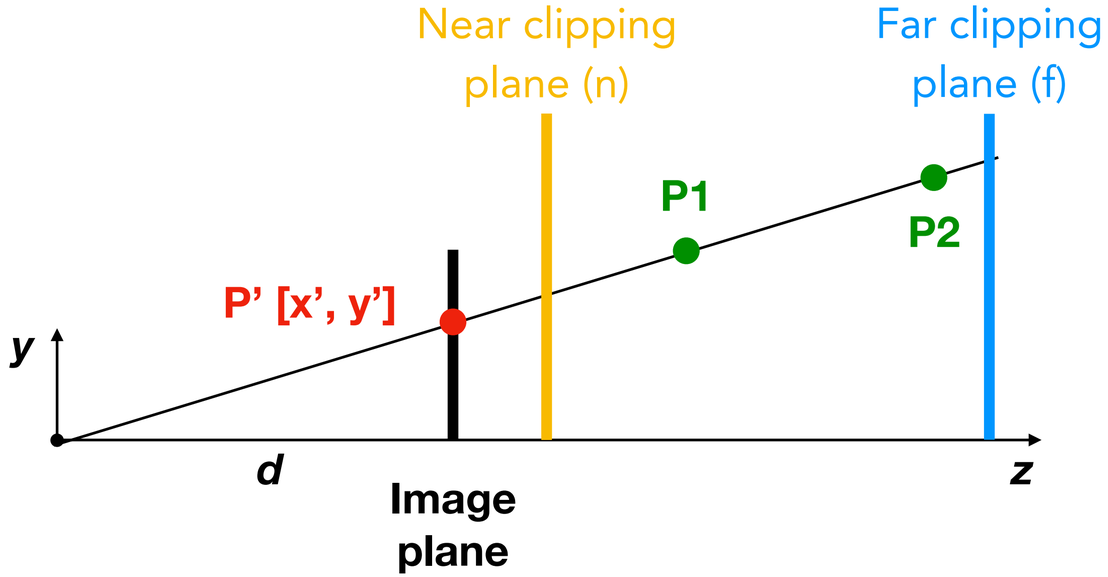
Same problem as before.

$$z'k = Czk = Cz^2 = xT_{00} + yT_{10} + zT_{20} + T_{30}$$

We need to ***bound z***.

Maintaining Z-Order: Idea

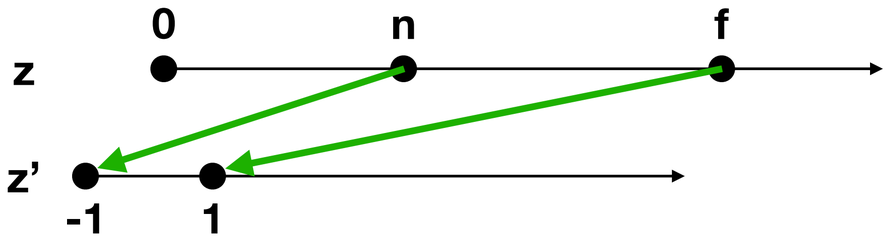




Idea: project the smallest z to 0 and largest z to 1 (or other fixed ranges).

- There is an *artificial* "near clipping plane" **n** and an *artificial* "far clipping" plane **f**.
- Only points between these two planes are visible to the camera.
- Image plane can be anywhere; technically not related to Near and Far clipping planes.

Maintaining Z-Order: Solution



$$z'k = z'z = xT_{02} + yT_{12} + zT_{22} + T_{32}$$

T_{02} and T_{12} must be 0

$$nT_{22} + T_{32} = -n$$

$$fT_{22} + T_{32} = f$$

$$\Rightarrow \quad T_{22} = (f+n)/(f-n), \quad T_{32} = -2fn/(f-n)$$

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} d & 0 & T_{02} & 0 \\ 0 & d & T_{12} & 0 \\ 0 & 0 & T_{22} & 1 \\ 0 & 0 & T_{32} & 0 \end{bmatrix} = \begin{bmatrix} x'k & y'k & z'k & k \end{bmatrix}$$

What About This Matrix?

$$[x \quad y \quad z \quad 1] \times \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = [xd \quad yd \quad 1 \quad z] \Longleftrightarrow [\frac{xd}{z} \quad \frac{yd}{z} \quad \frac{1}{z}]$$

The new z after transformation is inversely proportionally to depth. We don't need the near and far clipping planes any more.

- The visible region is no longer bounded.

This in theory is OK, but not used in practice:

- Numerical precision issue trickles in: 1/z could be too small or too large, exceeding digital number representation precision. No need to render objects too far anyways.

See CGPP, Chapter 13.3

Perspective Transformation Matrix (So Far)

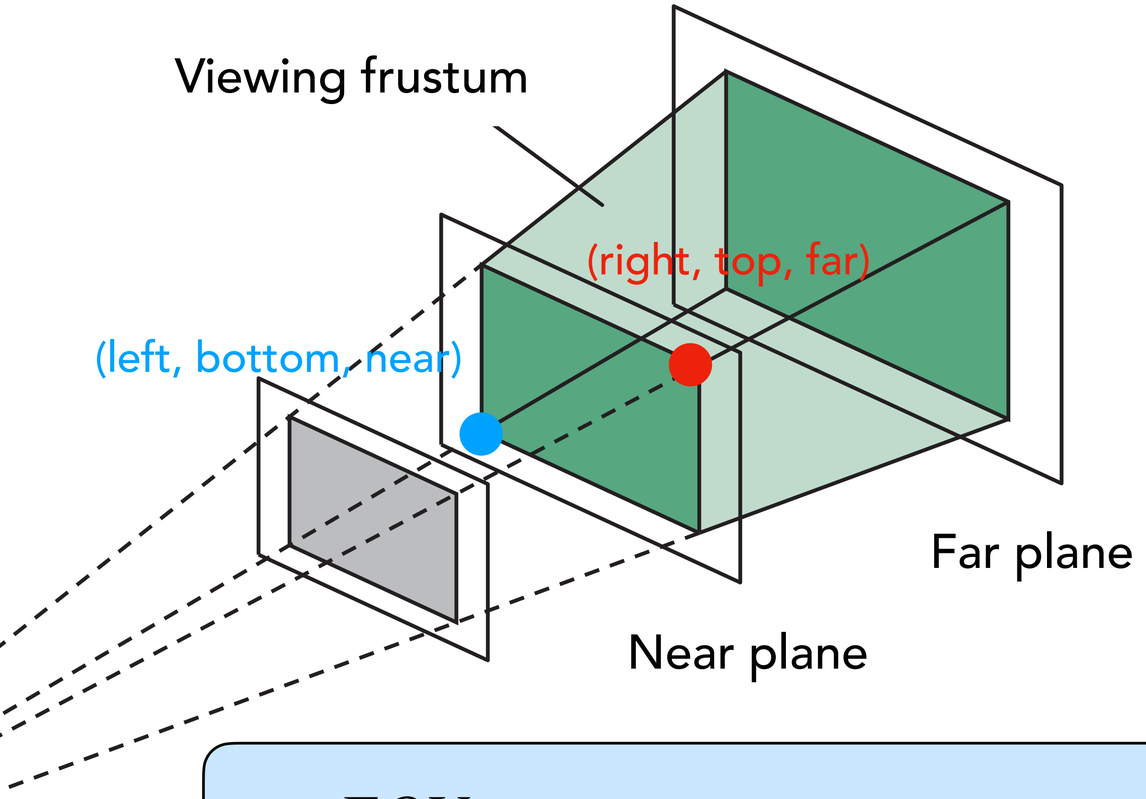
Perspective Projection $\begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2fn}{f-n} & 0 \end{bmatrix}$

Affine Transformation $\begin{bmatrix} T_{00} & T_{01} & T_{02} & 0 \\ T_{10} & T_{11} & T_{12} & 0 \\ T_{20} & T_{21} & T_{22} & 0 \\ T_{30} & T_{31} & T_{32} & 1 \end{bmatrix}$

Perspective projection:

- is **not** an affine transformation, which preserves line parallelisms.
- is a special case of **projective transformation** (a.k.a., **homography**), where all 16 coefficients can take arbitrary values (but only 15 free parameters/degrees of freedom because uniformly scaling all coefficients doesn't change the transformation)
- is not needed in/used by ray tracing.
- models only pinhole cameras (not enough to simulate depth of field, etc.)

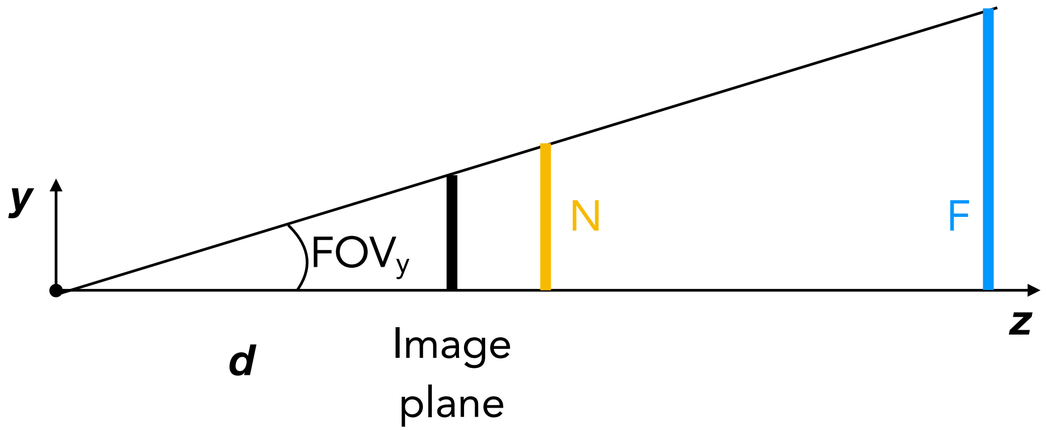
Viewing Frustum



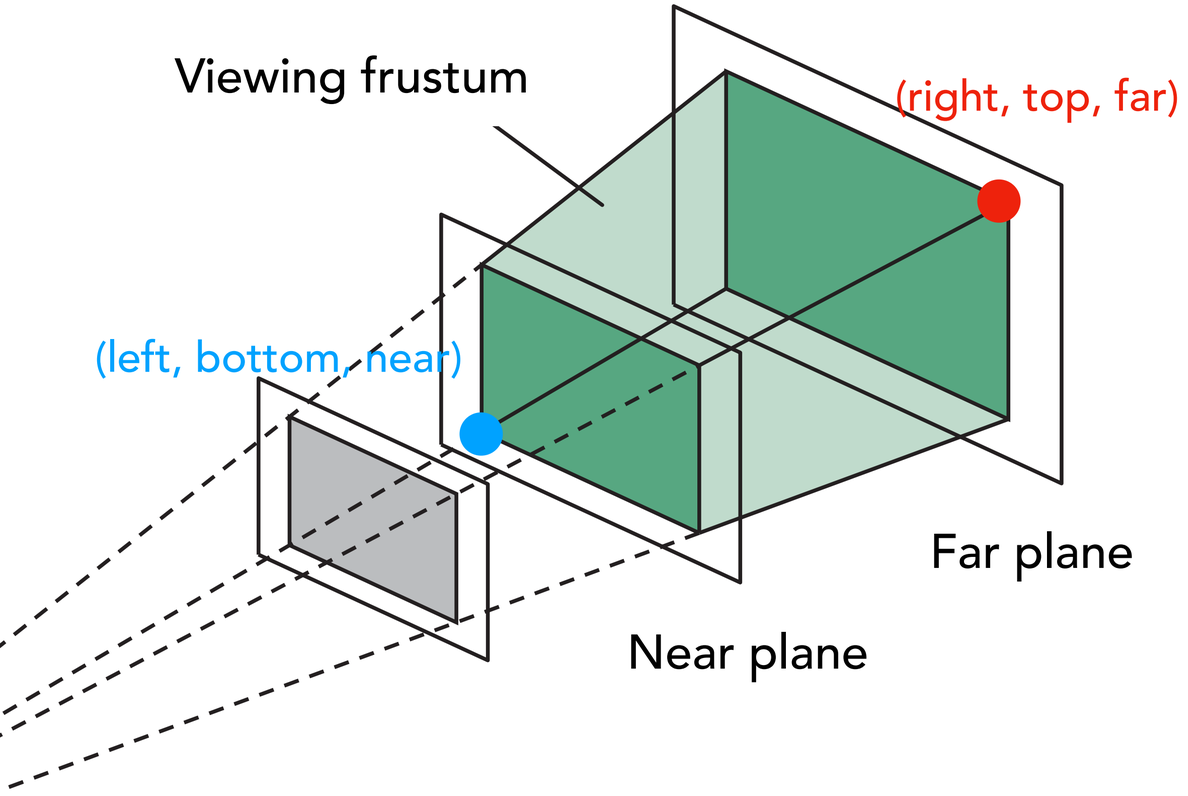
So far the visible part of the scene is clipped by the near and far planes.

But the visible region should also be bounded by the FOV (both horizontal and vertical) of the sensor.

$$\tan\frac{FOV_y}{2} = \frac{top}{near} \qquad \tan\frac{FOV_x}{2} = \frac{right}{near}$$



Viewing Frustum

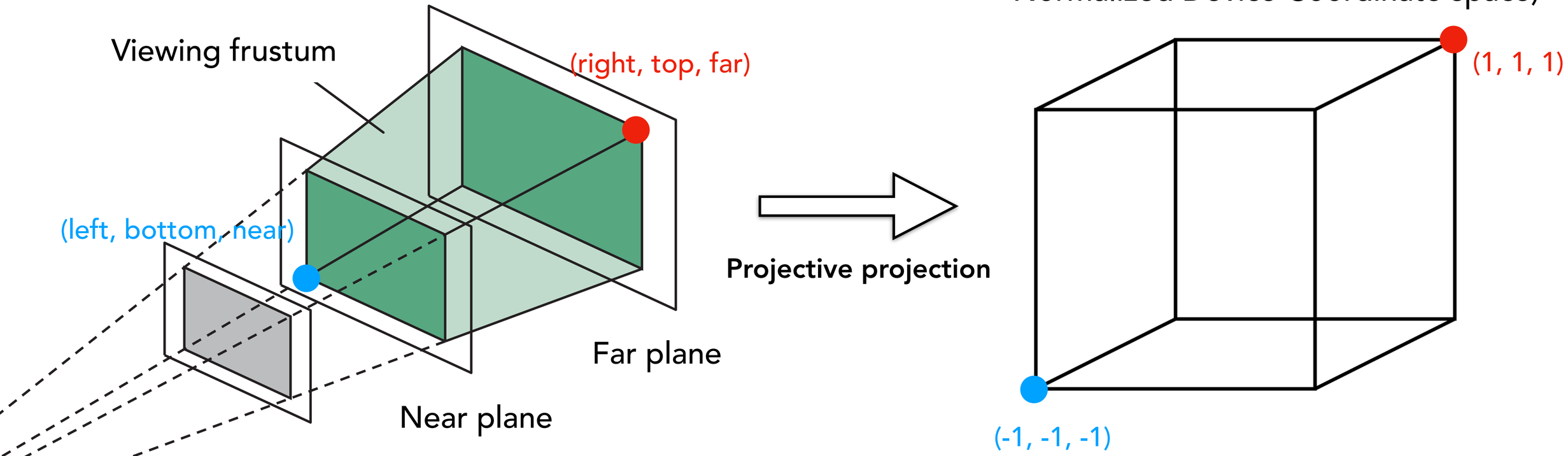


The visible part of the scene is actually a **frustum**.

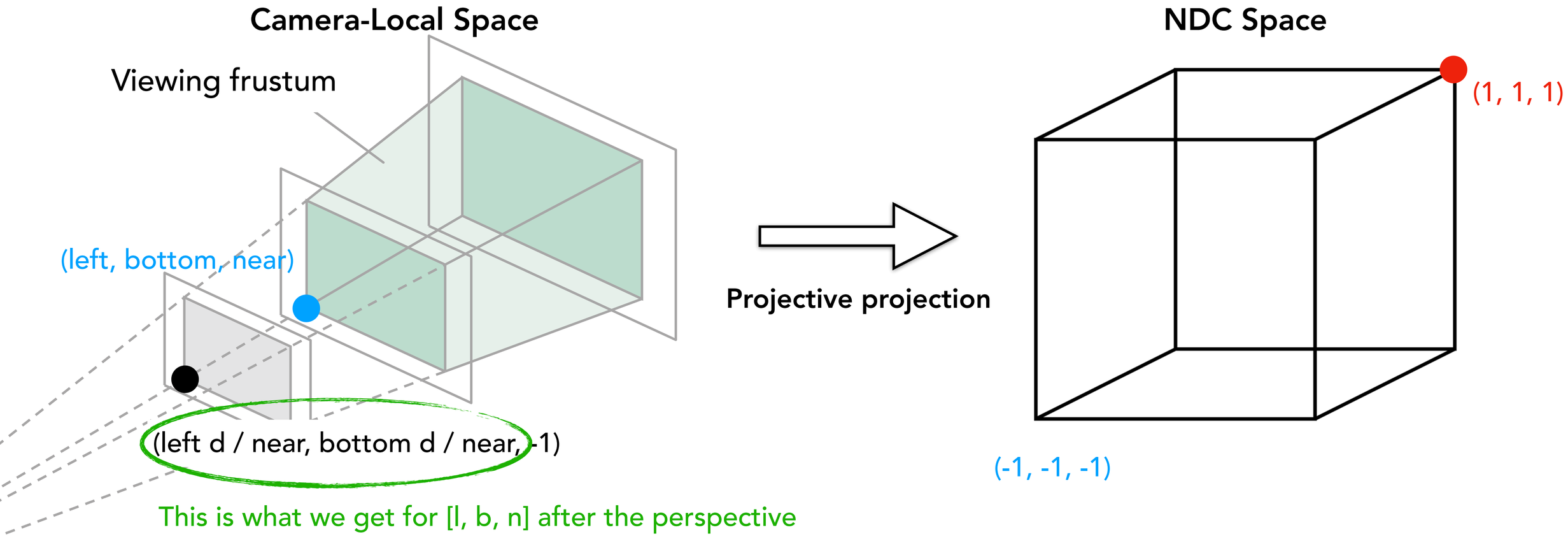
In rendering, we generally first map the frustum to a normalized cube that is independent of the actual sensor resolution.

- Then map the cube to the actual sensor resolution; in this way, any processing before that is decoupled from the sensor, which could change.

Viewing Frustum



Normalized Device Coordinate (NDC) Space



This is what we get for [l, b, n] after the perspective projection without being normalized to NDC. Before normalization, the frustum is projected to a hexahedron between [ld/n, bd/n, -1] and [rd/n, td/n, 1].

NDC Space (in XY Plane)

* The image plane need not be symmetric about the camera origin (pinhole), i.e., the sensor center is off the optical axis.

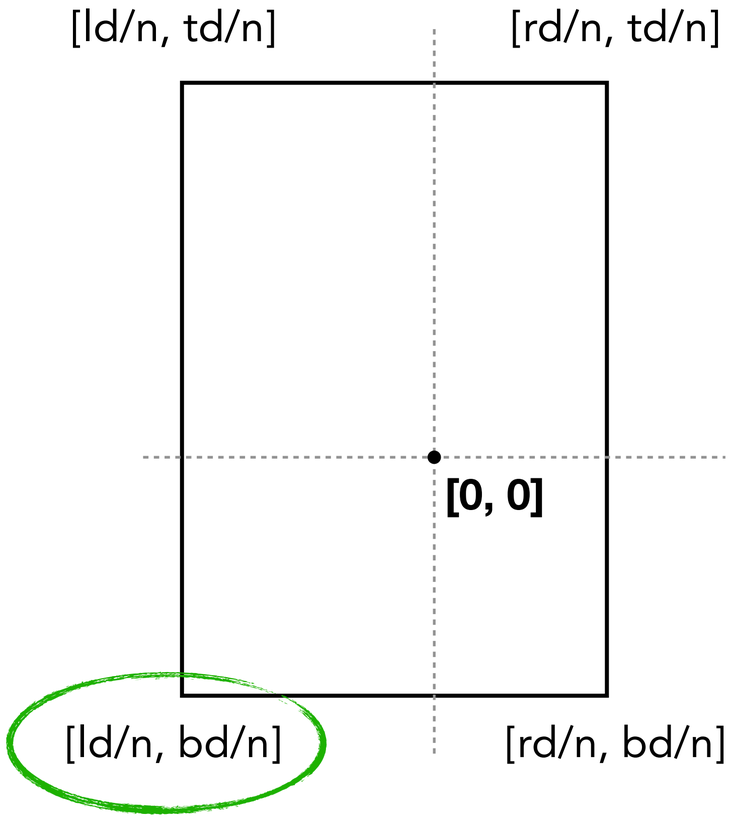
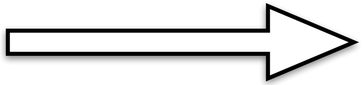
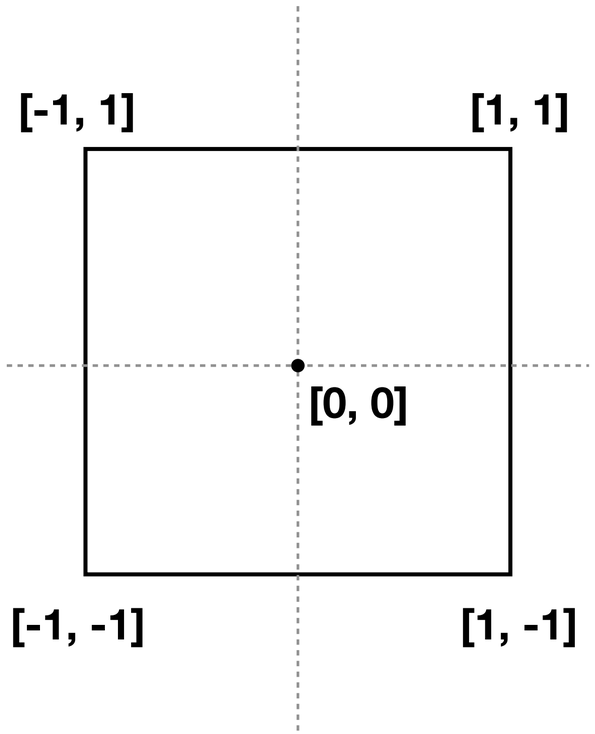


Image Plane (still in Camera Space)





NDC Space

NDC Space (in XY Plane)

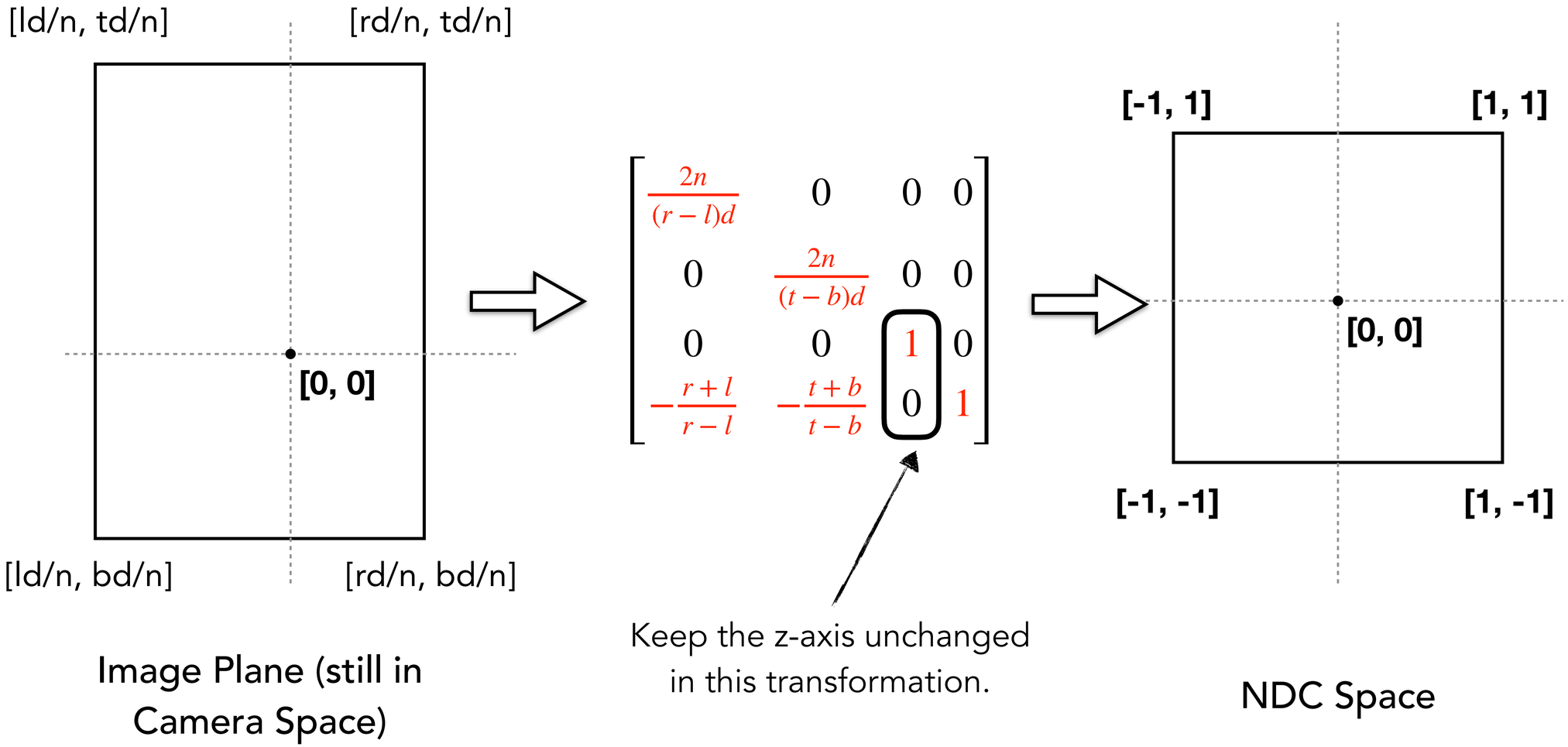


Image Plane (still in Camera Space)

Keep the z-axis unchanged in this transformation.

NDC Space

Overall Perspective Transformation

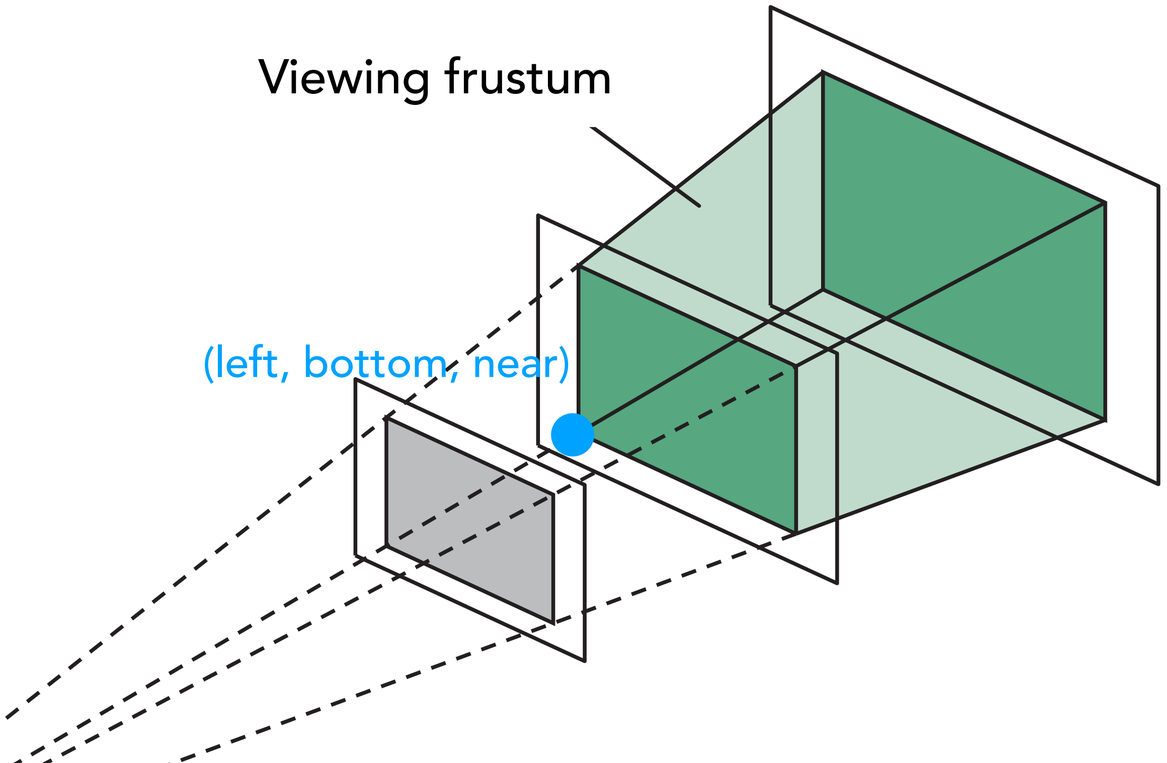
Bound the x and y axes within the FOV between [-1, 1]

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2fn}{f-n} & 0 \end{bmatrix} \times \begin{bmatrix} \frac{2n}{(r-l)d} & 0 & 0 & 0 \\ 0 & \frac{2n}{(t-b)d} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & 0 & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \times \begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2fn}{f-n} & 0 \end{bmatrix}$$

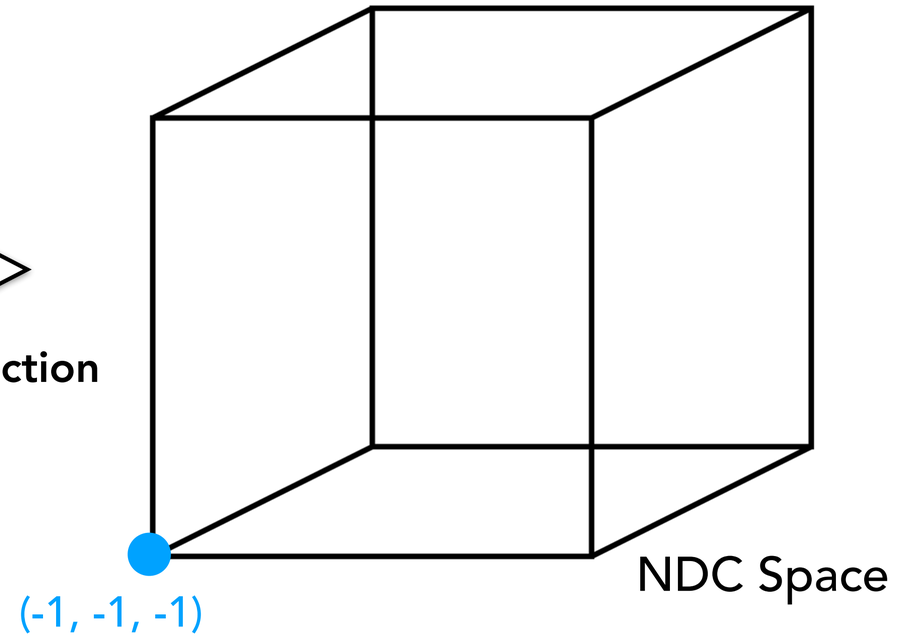
Perspective projection + bound the near and far clipping planes between [-1, 1] along z-axis

An Example

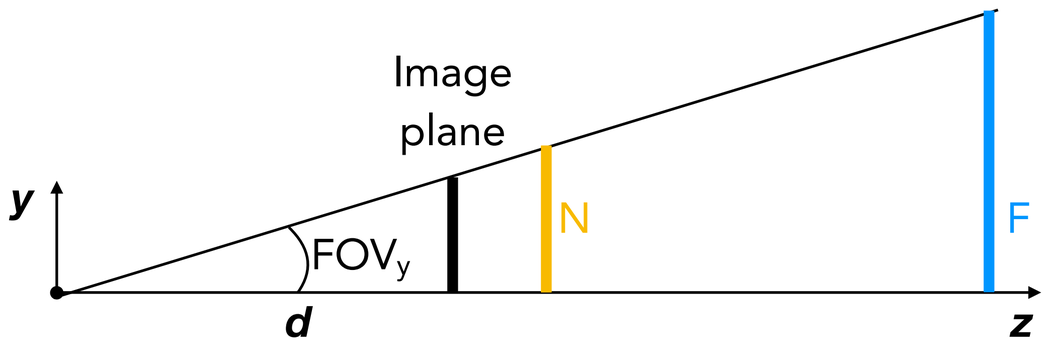
$$[l \quad b \quad n \quad 1] \times \begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2fn}{f-n} & 0 \end{bmatrix} = \begin{bmatrix} -n \\ -n \\ \frac{n(f+n)-2fn}{f-n} \\ n \end{bmatrix} = \begin{bmatrix} -n \\ -n \\ -n \\ n \end{bmatrix} \Longrightarrow \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$$



Perspective projection



The Matrix is Independent of Focal Length



$$\begin{bmatrix} \frac{2n}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2n}{t-b} & 0 & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2fn}{f-n} & 0 \end{bmatrix}$$

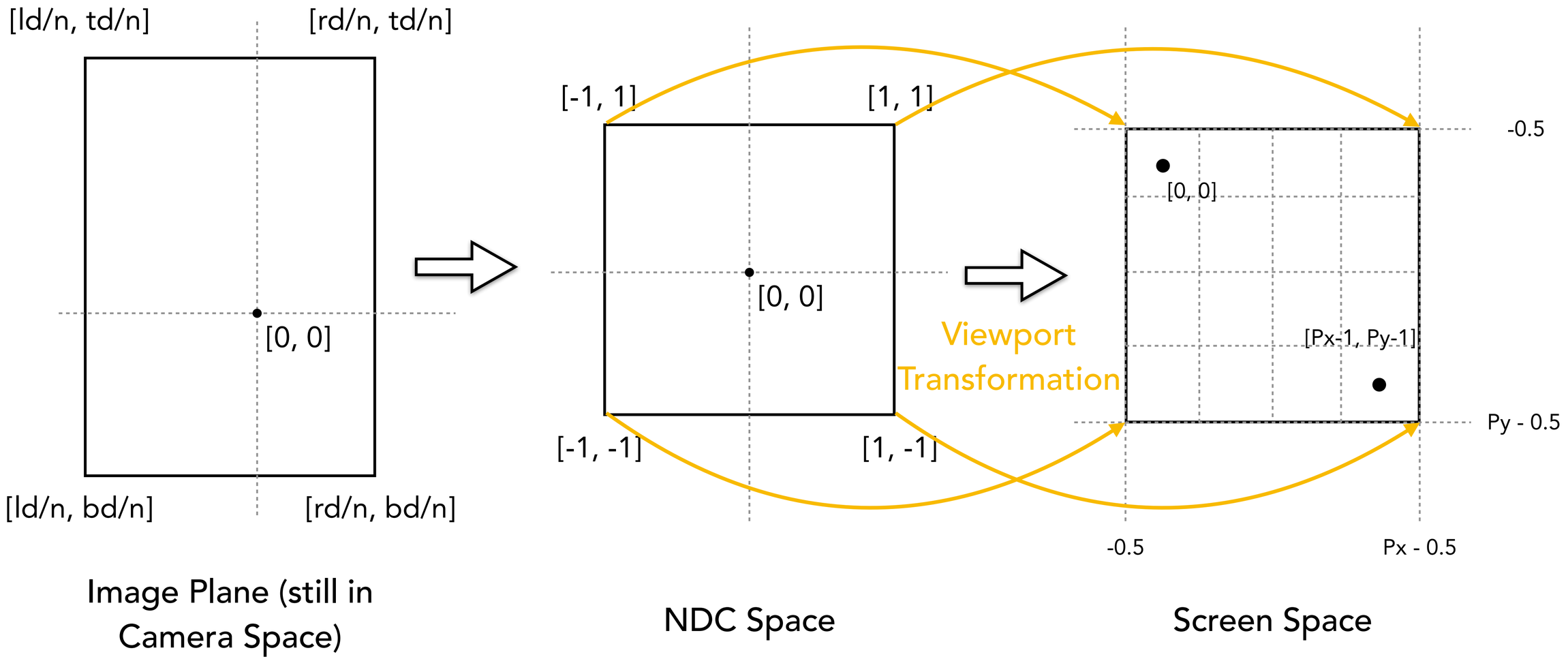
The perspective matrix is completely independent of the focal length **d**.

- It does depend on **r**, **l**, **t**, **d**, **n**, **f**, which uniquely define a frustum.
- **r**, **l**, **t**, **d**, **n**, **f** are related by the FOV (x and y) of the sensor.

Because the matrix transforms the visible region of the scene to a normalized cube, and given a FOV, what's visible to the camera is fixed, i.e., the frustum.

- In OpenGL/WebGL, the near clipping plane is placed at the focal length so that d never shows up during the derivation, but that's unnecessary and a bit confusing.

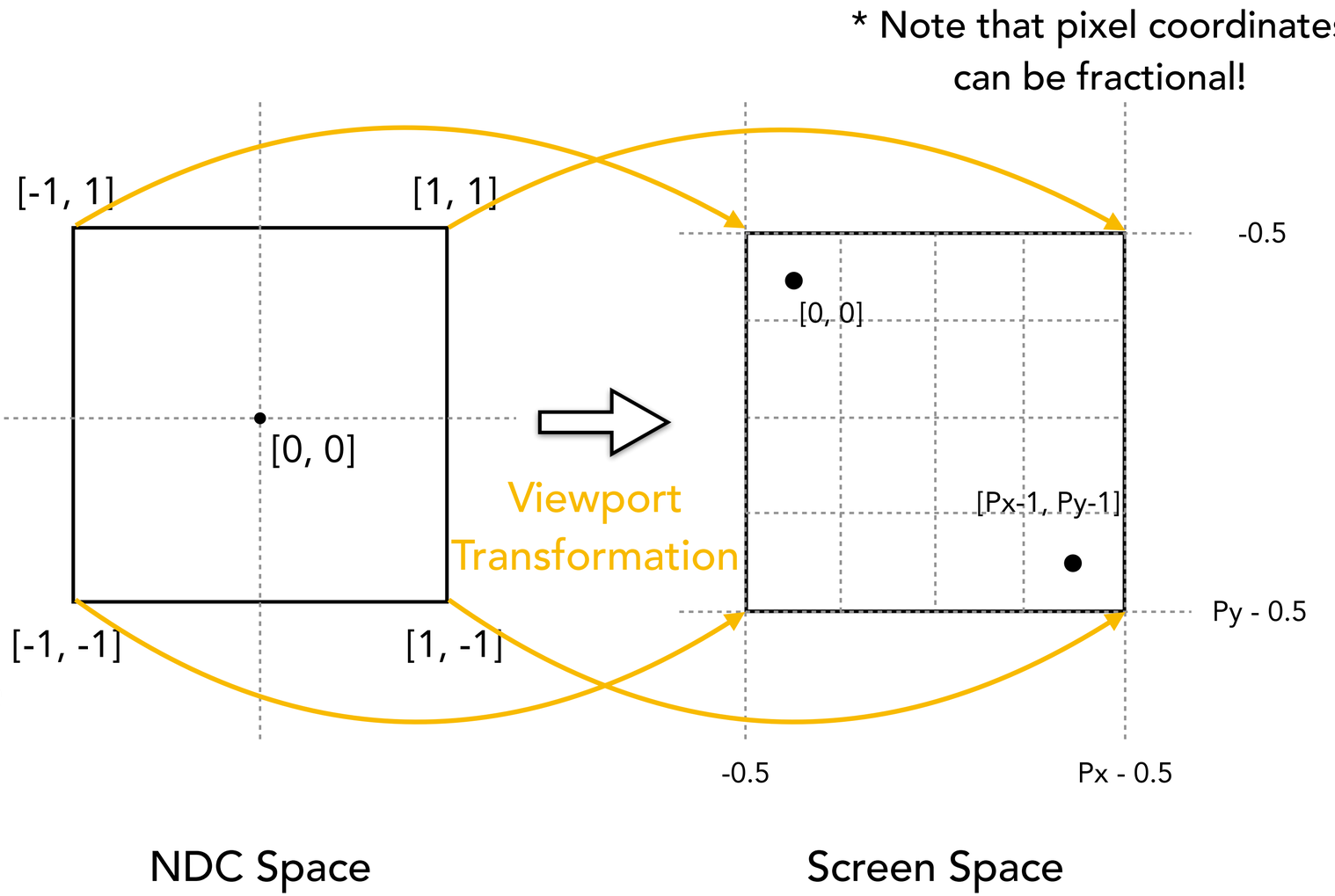
Generating Pixel Coordinates in Screen Space

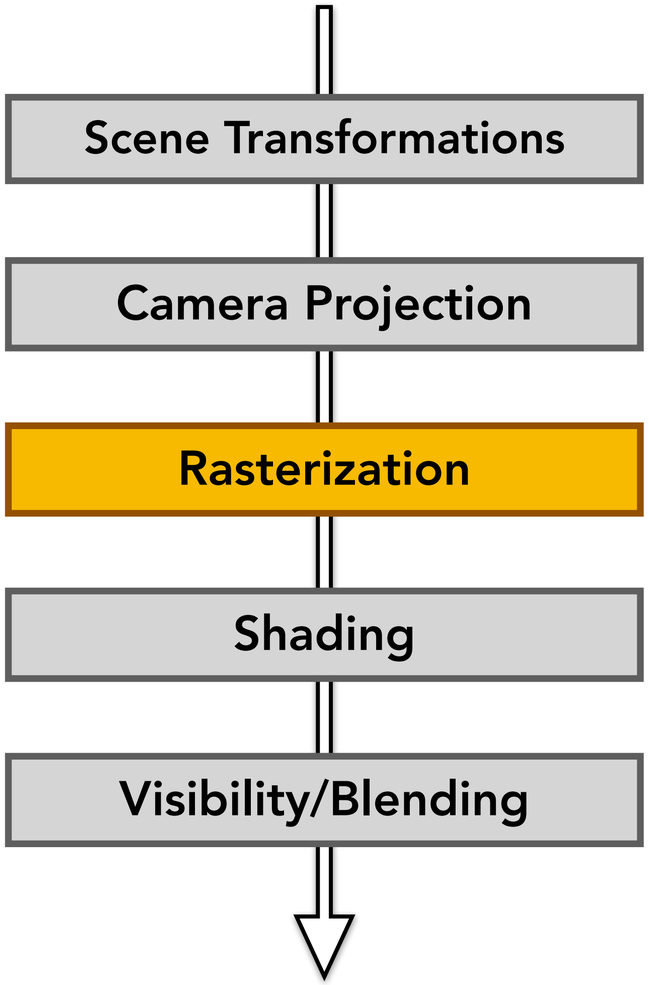


Notes on Screen Space

Convention: the origin of the screen space is the center of the top-left pixel.

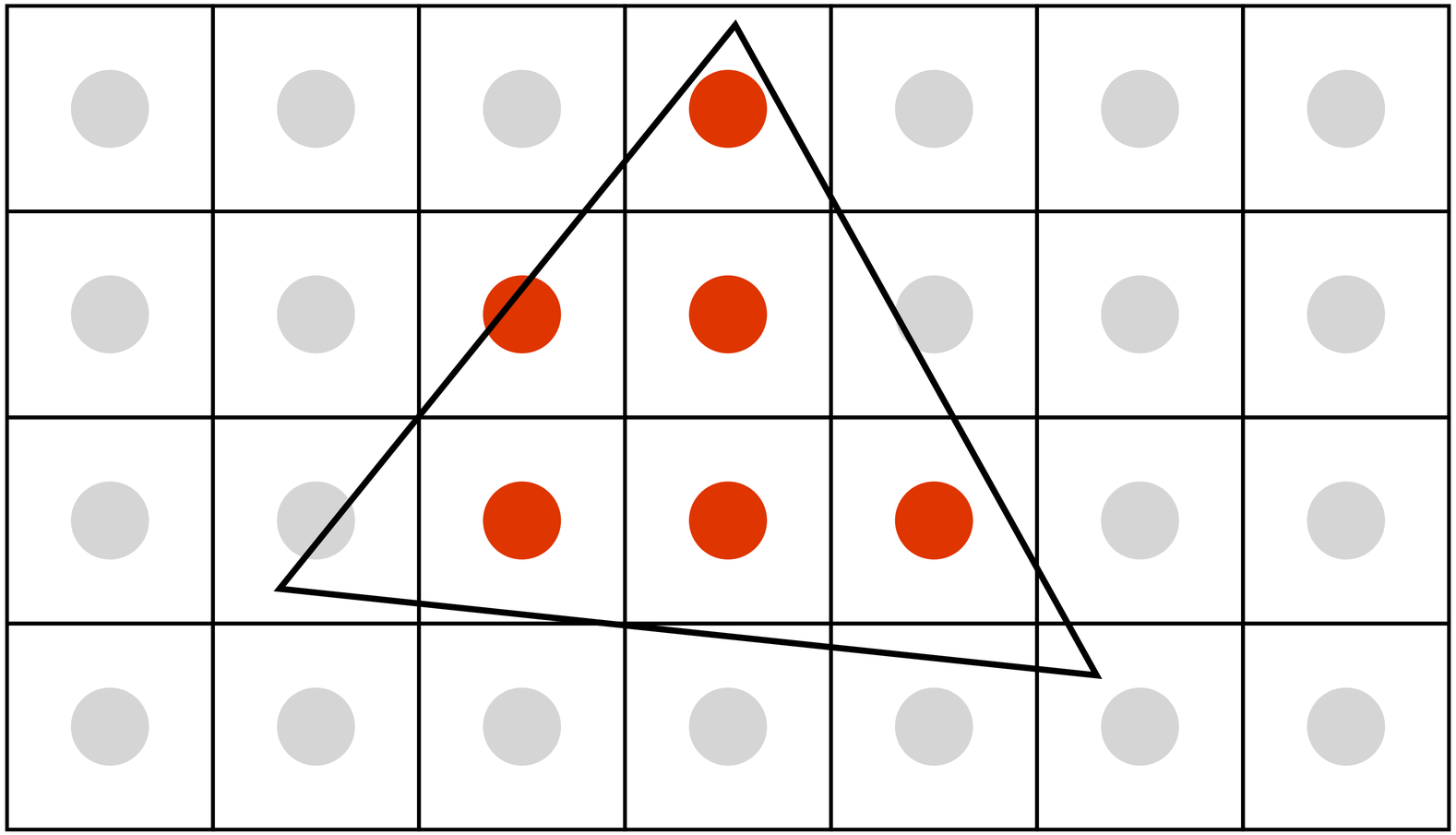
The screen space is still **continuous**. That is, pixel coordinates can be **fractional**! Later we will "**rasterize**" the screen space to generate actual pixels at integer coordinates.



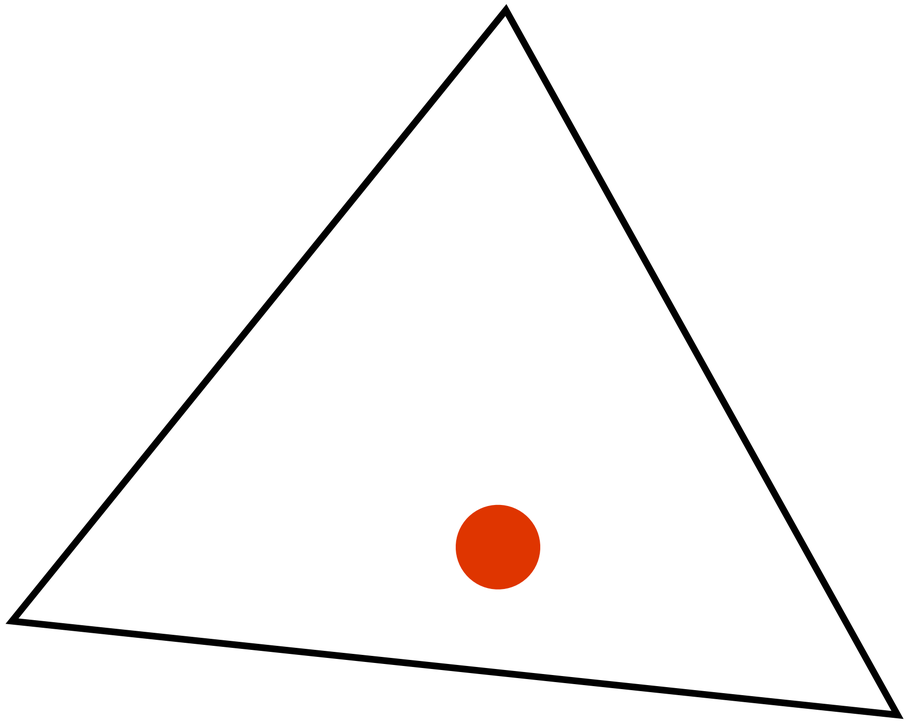


Rasterization

Which Pixels are Covered by Each Triangle?



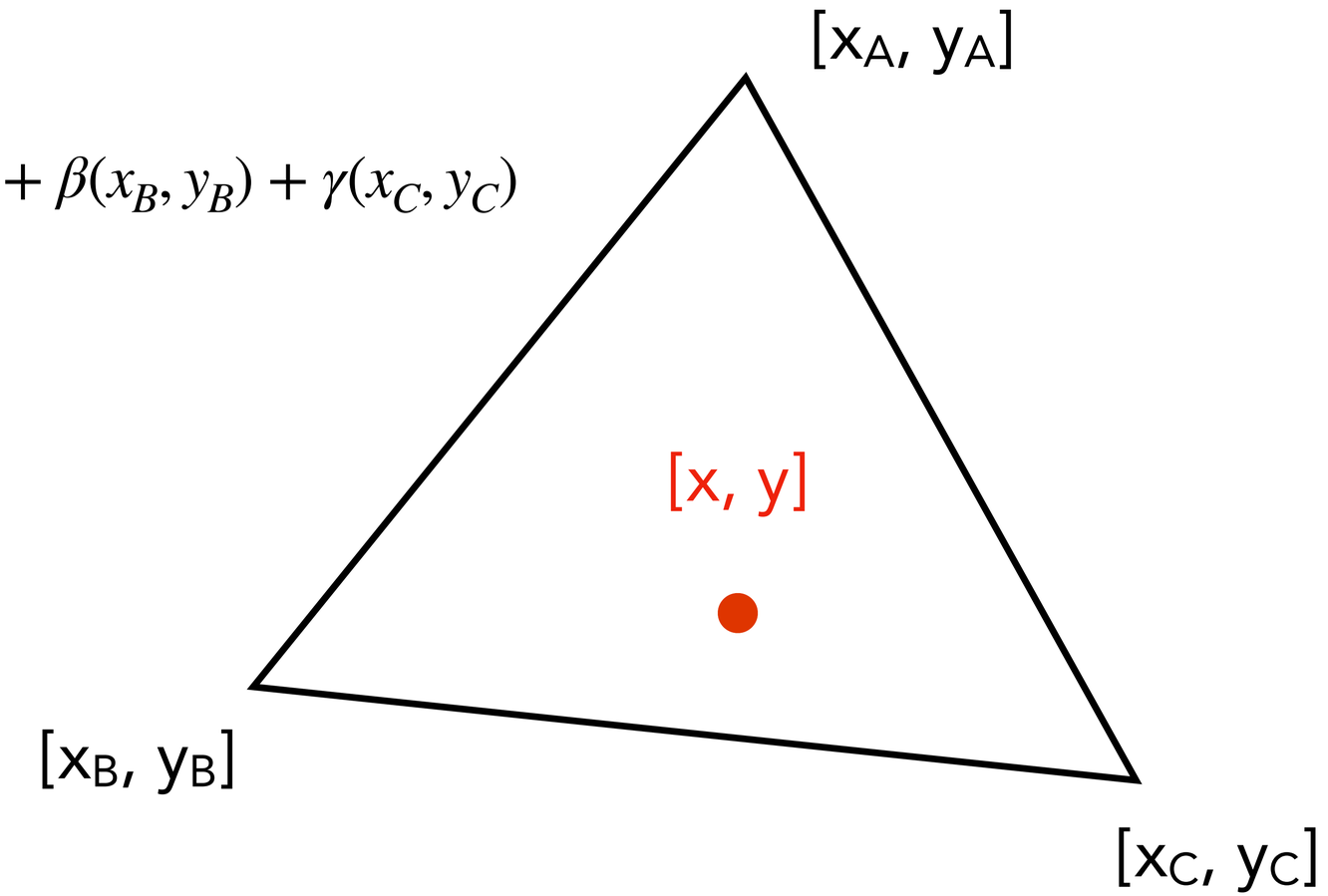
Key Question: Is a Point Inside a Triangle?



Barycentric Coordinates

$$(x, y) = \alpha(x_A, y_A) + \beta(x_B, y_B) + \gamma(x_C, y_C)$$

$$\alpha + \beta + \gamma = 1$$



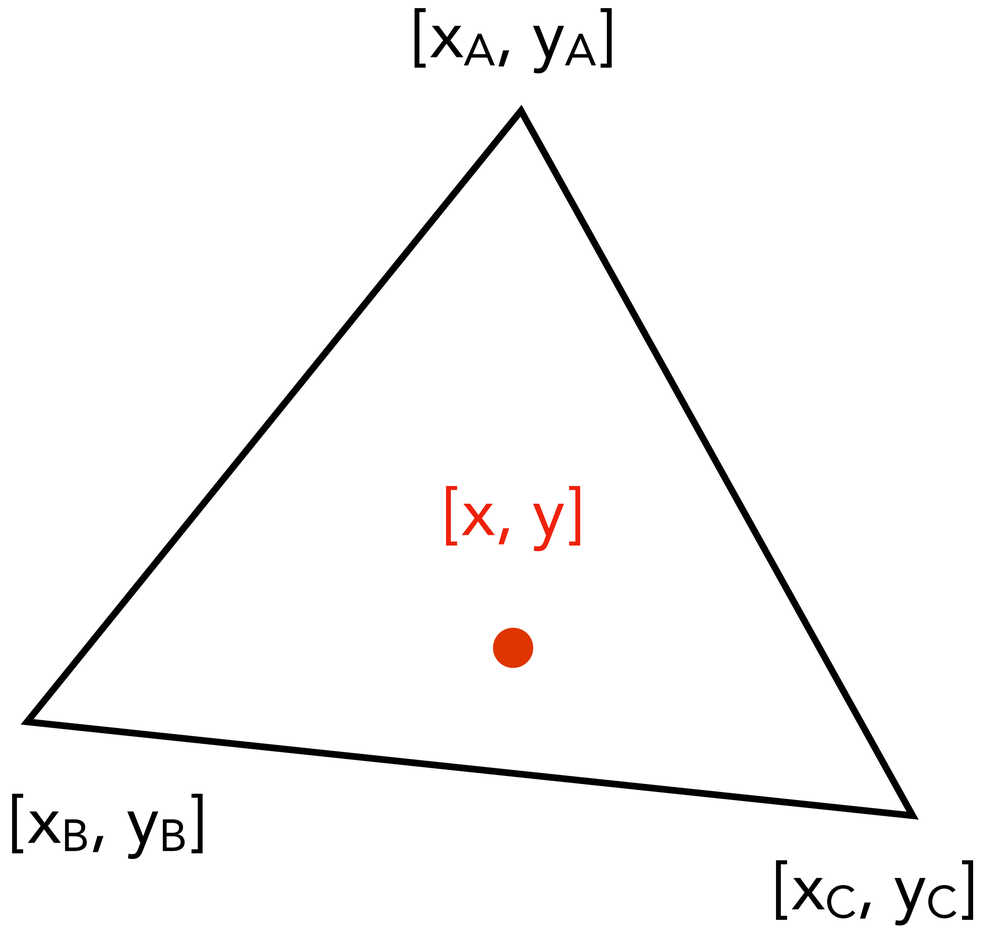
Barycentric Coordinates

$$(x, y) = \alpha(x_A, y_A) + \beta(x_B, y_B) + \gamma(x_C, y_C)$$

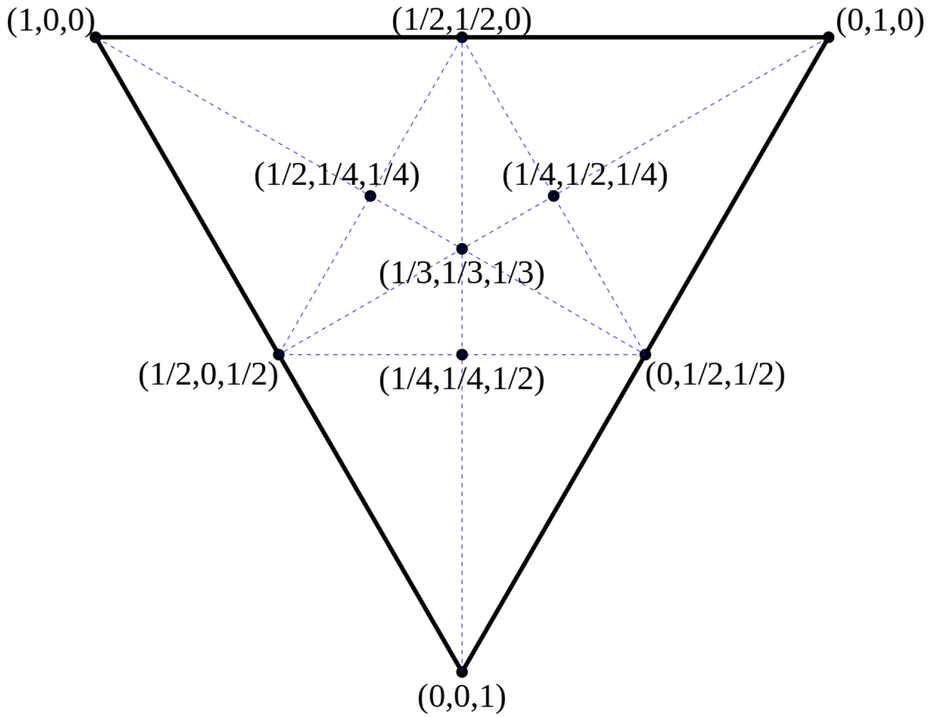
$$\alpha + \beta + \gamma = 1$$

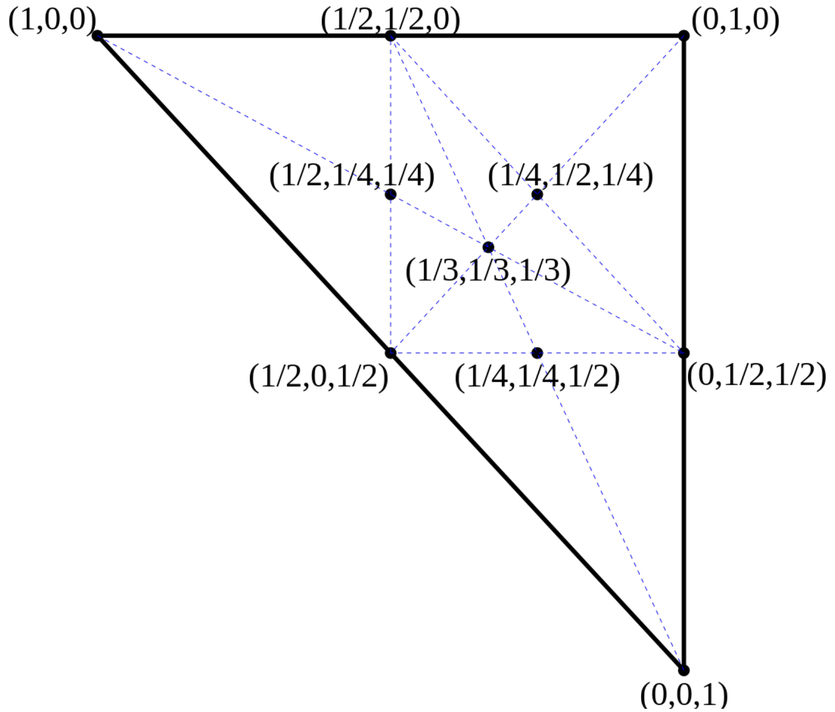
$$\alpha = \frac{-(x - x_B)(y_C - y_B) + (y - y_B)(x_C - x_B)}{-(x_A - x_B)(y_C - y_B) + (y_A - y_B)(x_C - x_B)}$$

$$\beta = \frac{-(x - x_C)(y_A - y_C) + (y - y_C)(x_A - x_C)}{-(x_B - x_C)(y_A - y_C) + (y_B - y_C)(x_A - x_C)}$$



Barycentric Coordinates Examples





https://en.wikipedia.org/wiki/Barycentric_coordinate_system

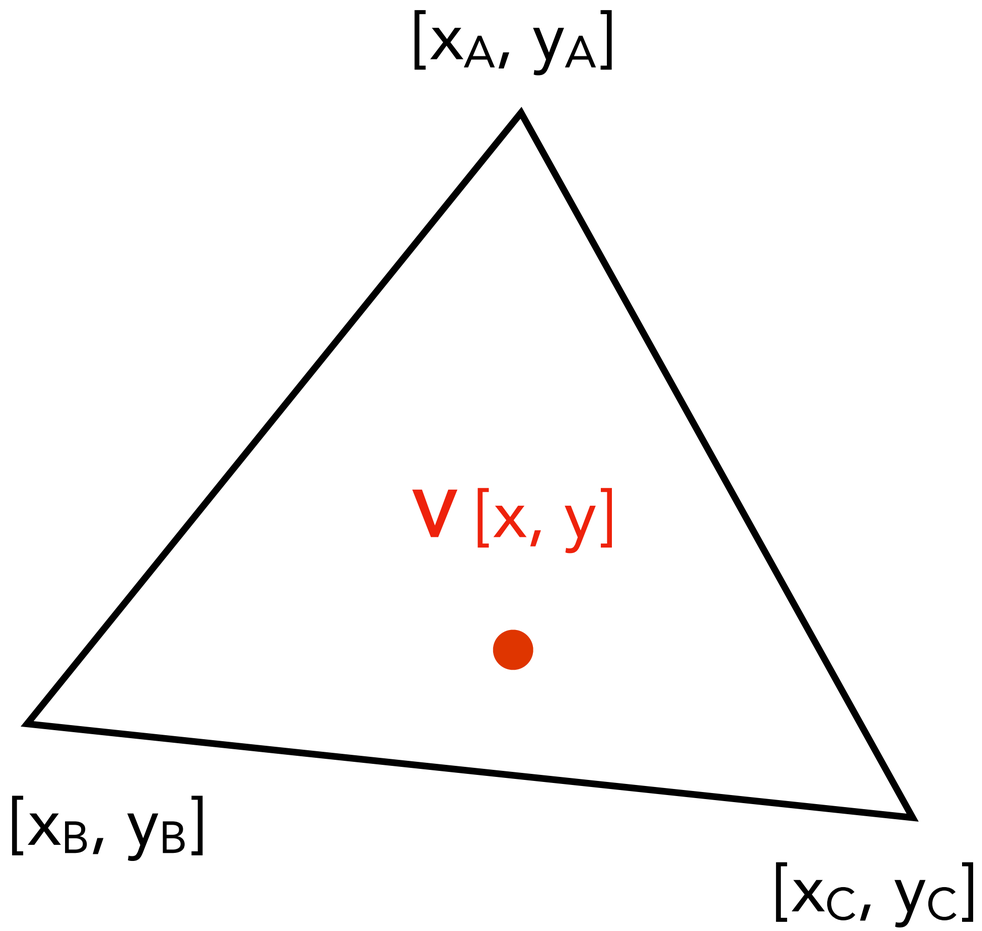
Point in Triangle Test

$$(x, y) = \alpha(x_A, y_A) + \beta(x_B, y_B) + \gamma(x_C, y_C)$$

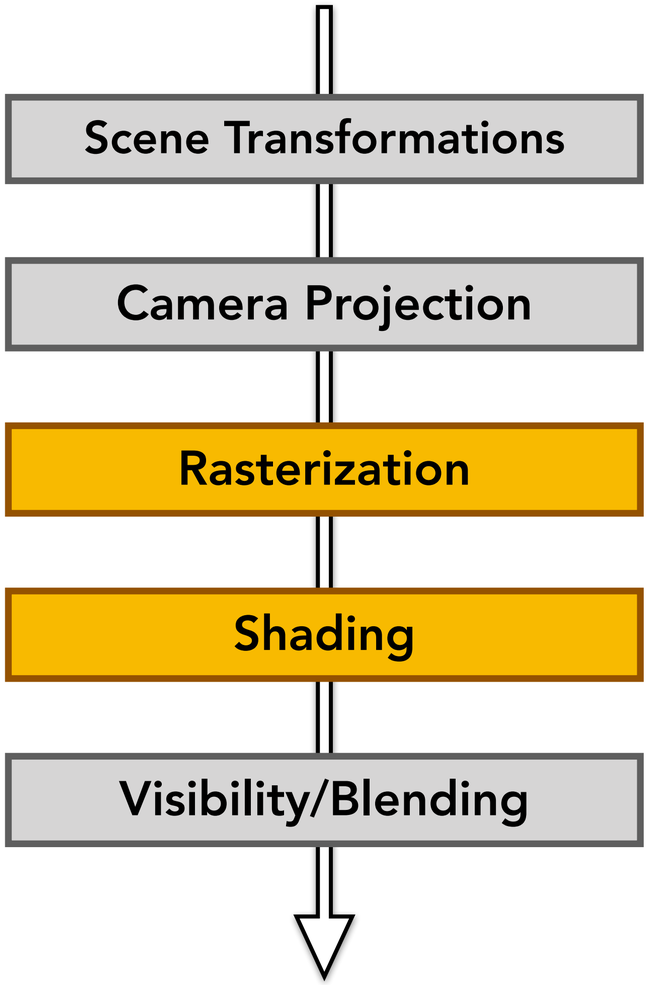
$$\alpha + \beta + \gamma = 1$$

For any V that's inside the triangle:
0 <= α, β, γ <= 1

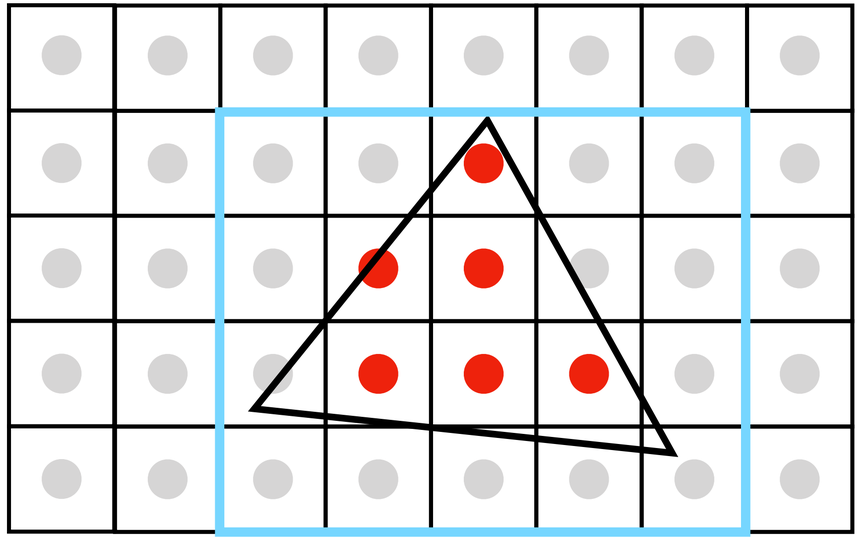
For any V that's outside the triangle:
Some of α, β, γ is outside the [0, 1] range.



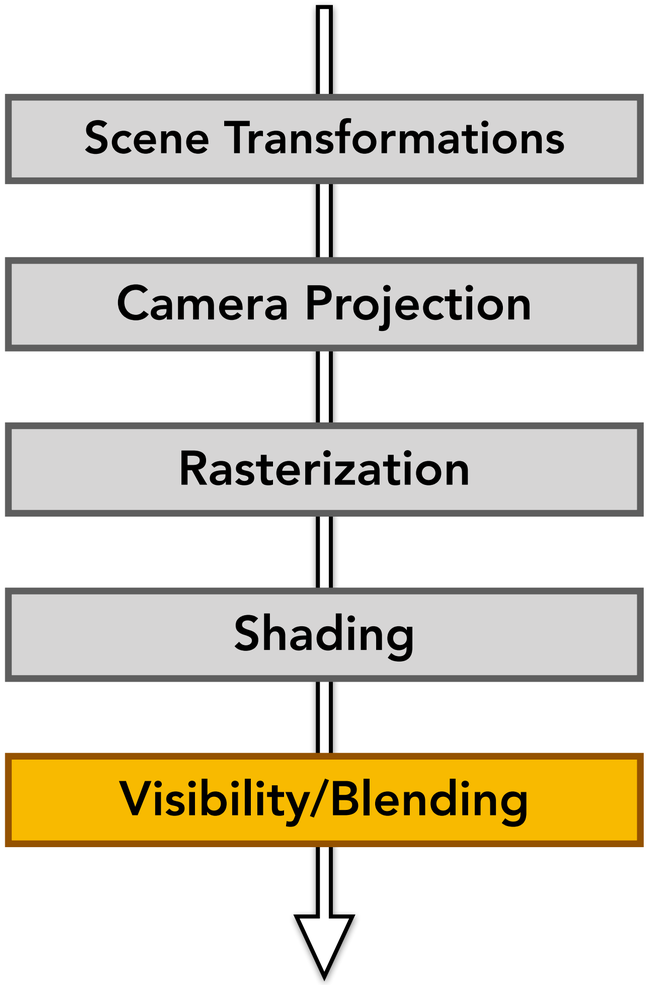
Rasterization Algorithm (w/ Simple Shading)




```
Foreach triangle in mesh
  Perspective project triangle to canvas;
  Foreach pixel in image
    if (pixel is in the projected triangle)
      pixel.color = triangle.color; // shading
```



Could first find the bounding box of the triangle to narrow the search space.

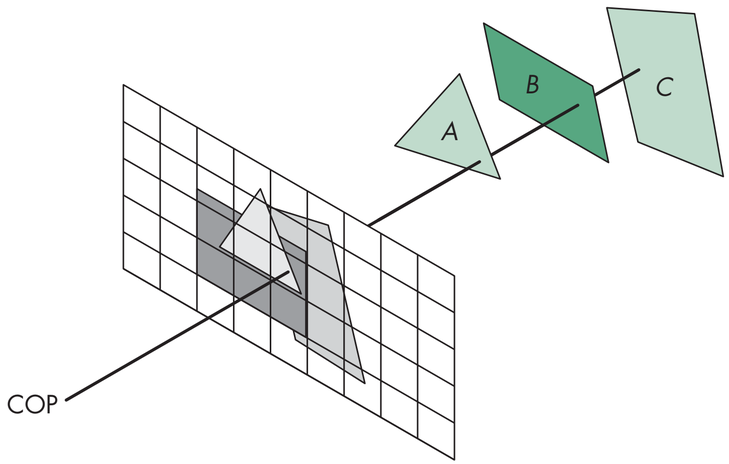


Visibility and Blending

Visibility (Hidden Surface) Problem

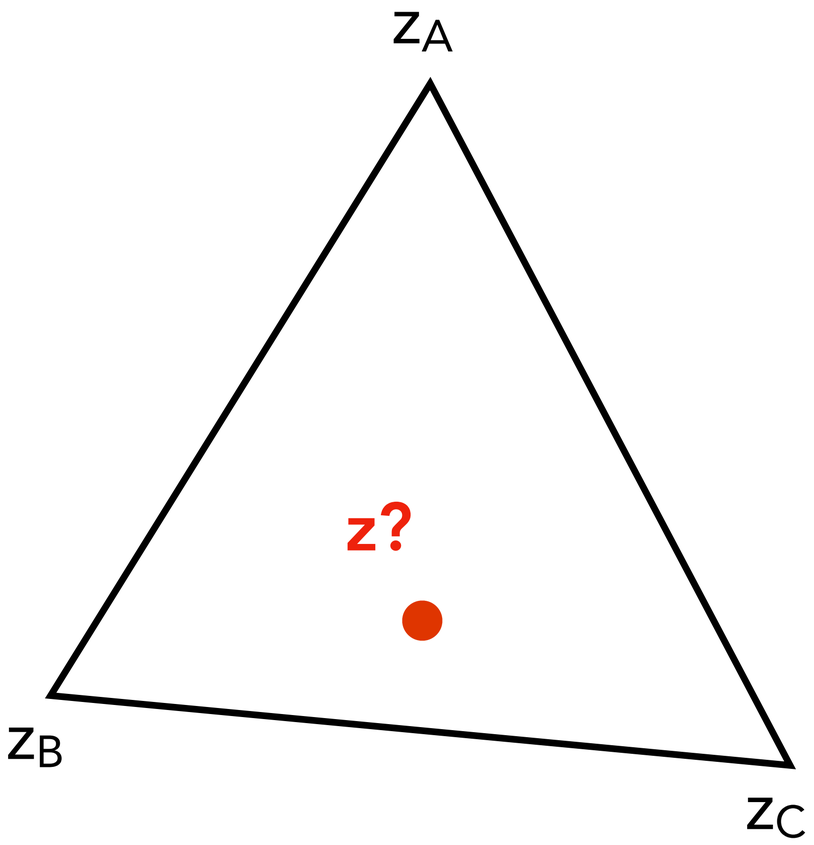
When multiple points in the scene get projected to the same pixel, must determine which point "wins", i.e., gets to assign its color to the pixel.

Fortunately, perspective projection maintains the relative point depth. Determining the relative depth is done using a **depth-buffer** or a z-buffer.




```
Foreach triangle in mesh
  Perspective project triangle to canvas;
  Foreach pixel in image
    if (pixel is in the projected triangle)
      D = computeDepth(pixel)
      if (D < depthBuffer[pixel])
        shade(pixel)
        depthBuffer[pixel] = D
```

Calculating Depth

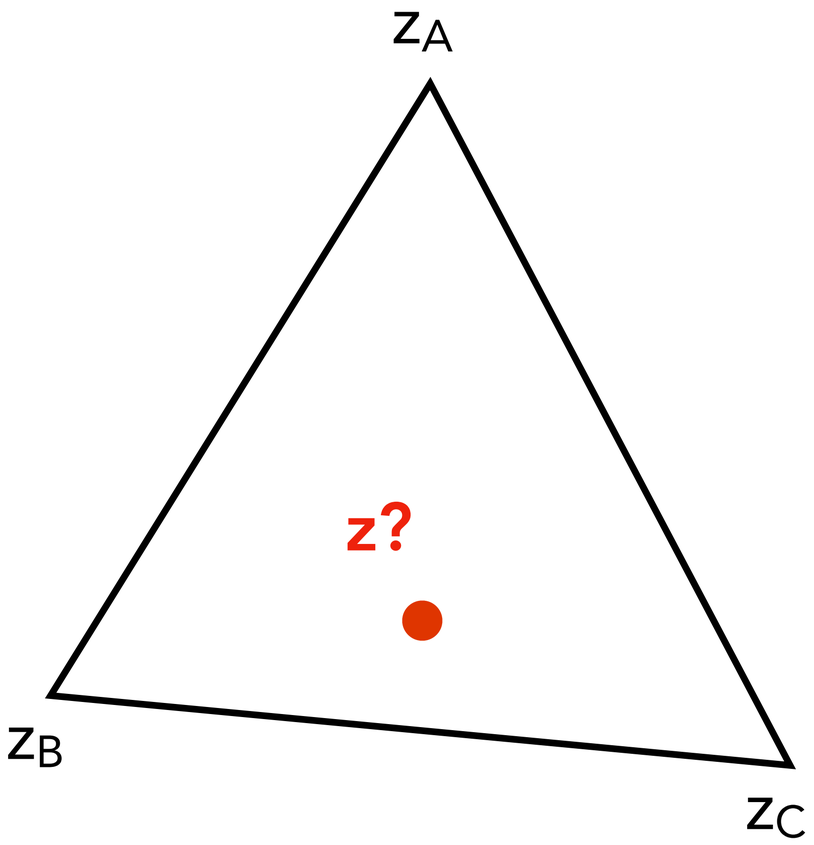


We know the depths (z-axis) of triangle vertices (inverting the perspective matrix).

How about other pixels? Can we interpolate based on barycentric coordinates?

```
Foreach triangle in mesh
  Perspective project triangle to canvas;
  Foreach pixel in image
    if (pixel is in the projected triangle)
      D = computeDepth(pixel)
      if (D < depthBuffer[pixel])
        shade(pixel)
        depthBuffer[pixel] = D
```

Calculating Depth

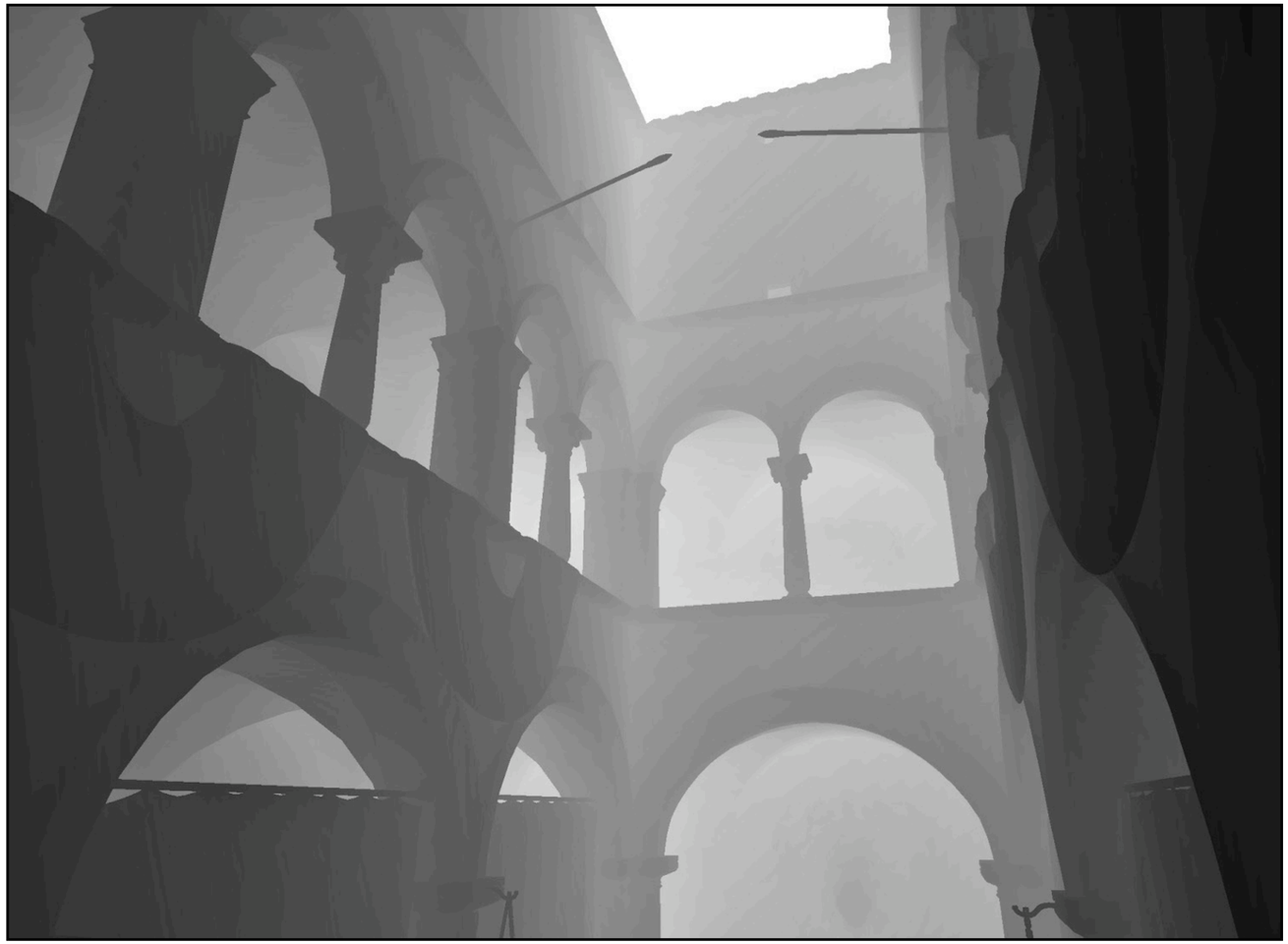


We know the depths (z-axis) of triangle vertices (inverting the perspective matrix).

How about other pixels? Can we interpolate based on barycentric coordinates?

Yes, but the barycentric coordinates need to be calculated in the camera space (3D), not in the screen space (2D)!

Visualizing Depth Map



https://forum.unity.com/threads/how-to-manually-write-to-depth-buffer-before-post-effects.528243/ http://glampert.com/2014/01-26/visualizing-the-depth-buffer/

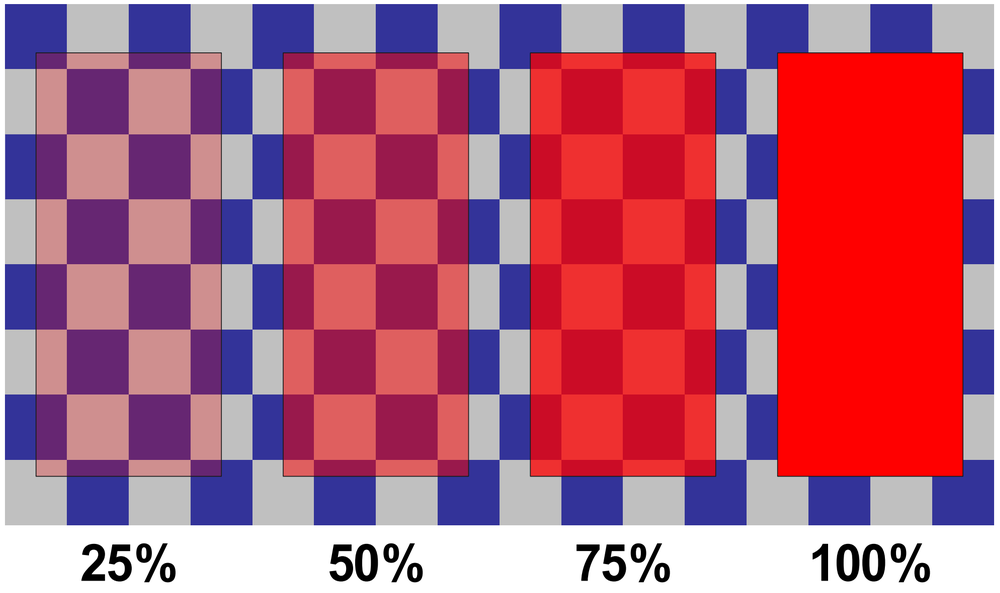
Alpha Blending

We can also simulate transparent materials by blending colors from different primitives when they map to the same pixel.

- Use an **alpha** channel to represent opacity.

This is purely a hack. Not physically based. Remember how to properly simulate transparency?

- Will revisit this later.



Color = **alpha** x Foreground Color + (1 - **alpha**) * Background Color

Aliasing and Anti-Aliasing in Shading

Simple Shading

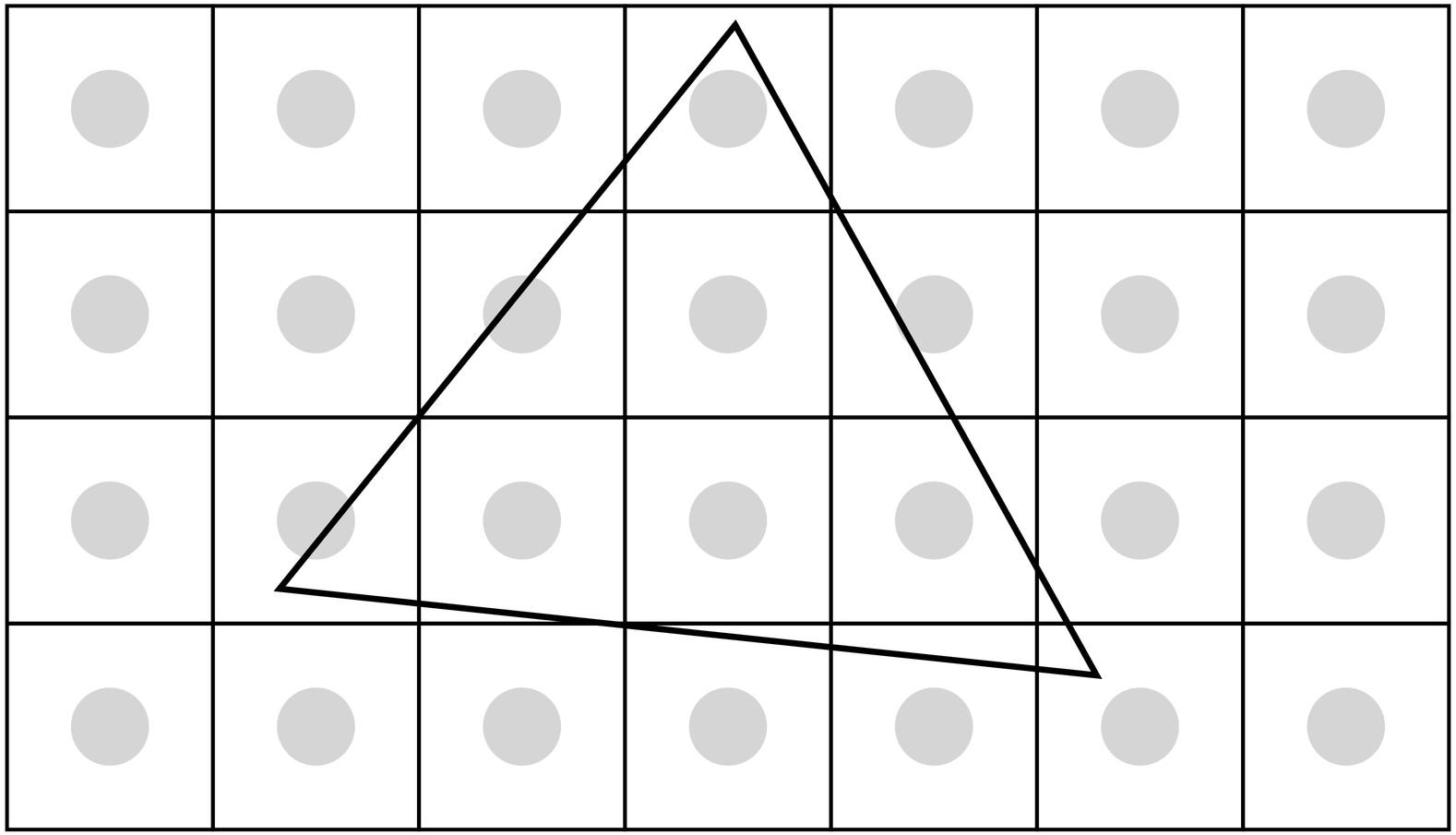
Basic assumption: each triangle face is assigned a color.

- Called “per-face shading”.
- …or each triangle vertex has a color, and color of any point inside the triangle is interpolated (per-vertex shading).
- …or each point’s color is calculated by incident lights and viewing angle (per-fragment shading); can be empirical or physically-based.
- We will talk about more realistic shading later, but the general idea here applies.

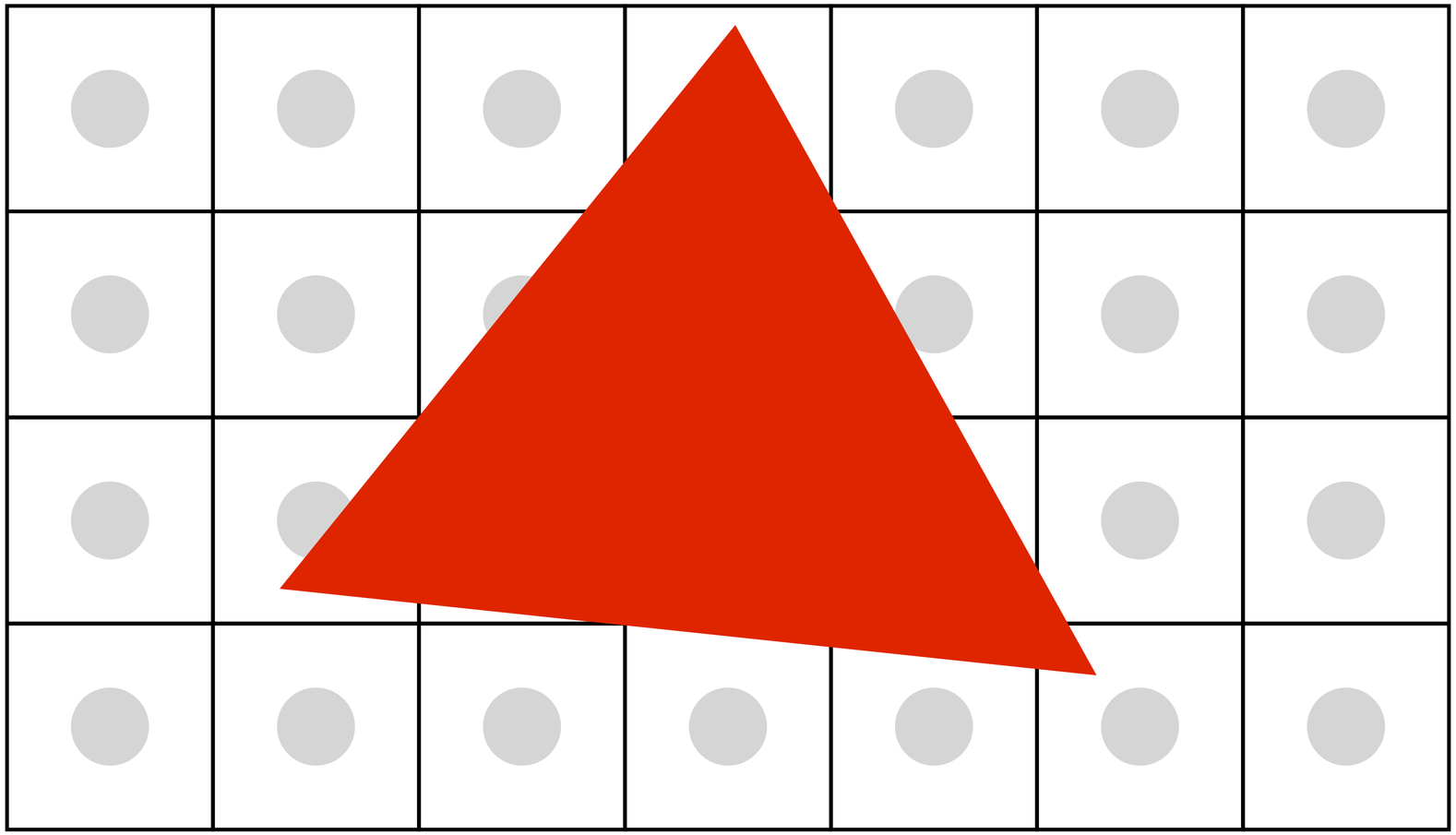
Question: how to assign color to each pixel?

- Simple? If a pixel is inside a triangle, it gets the triangle color.
- Issue: a pixel is a continuous spatial region, not just a point on triangle.

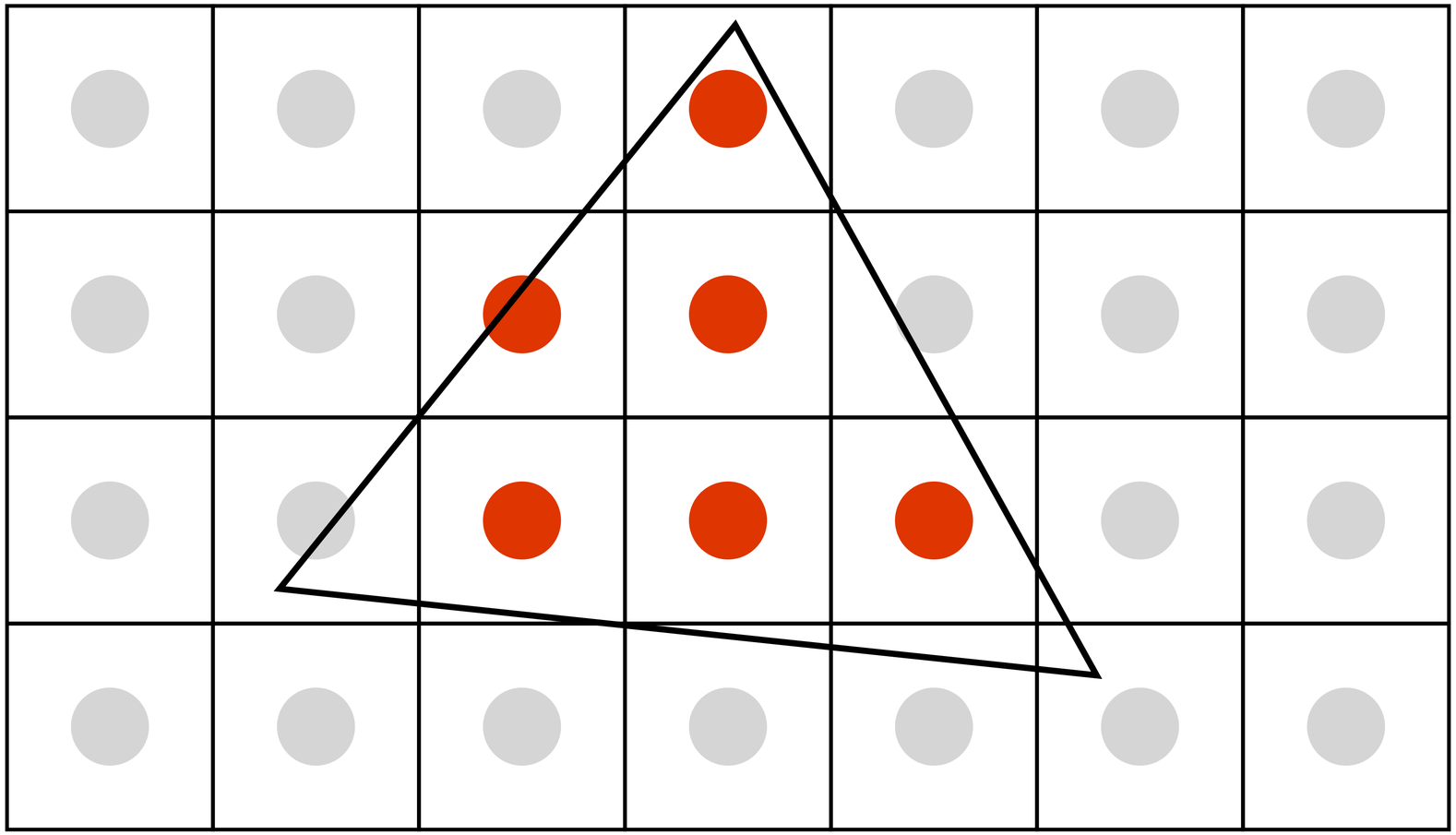
Simple Shading



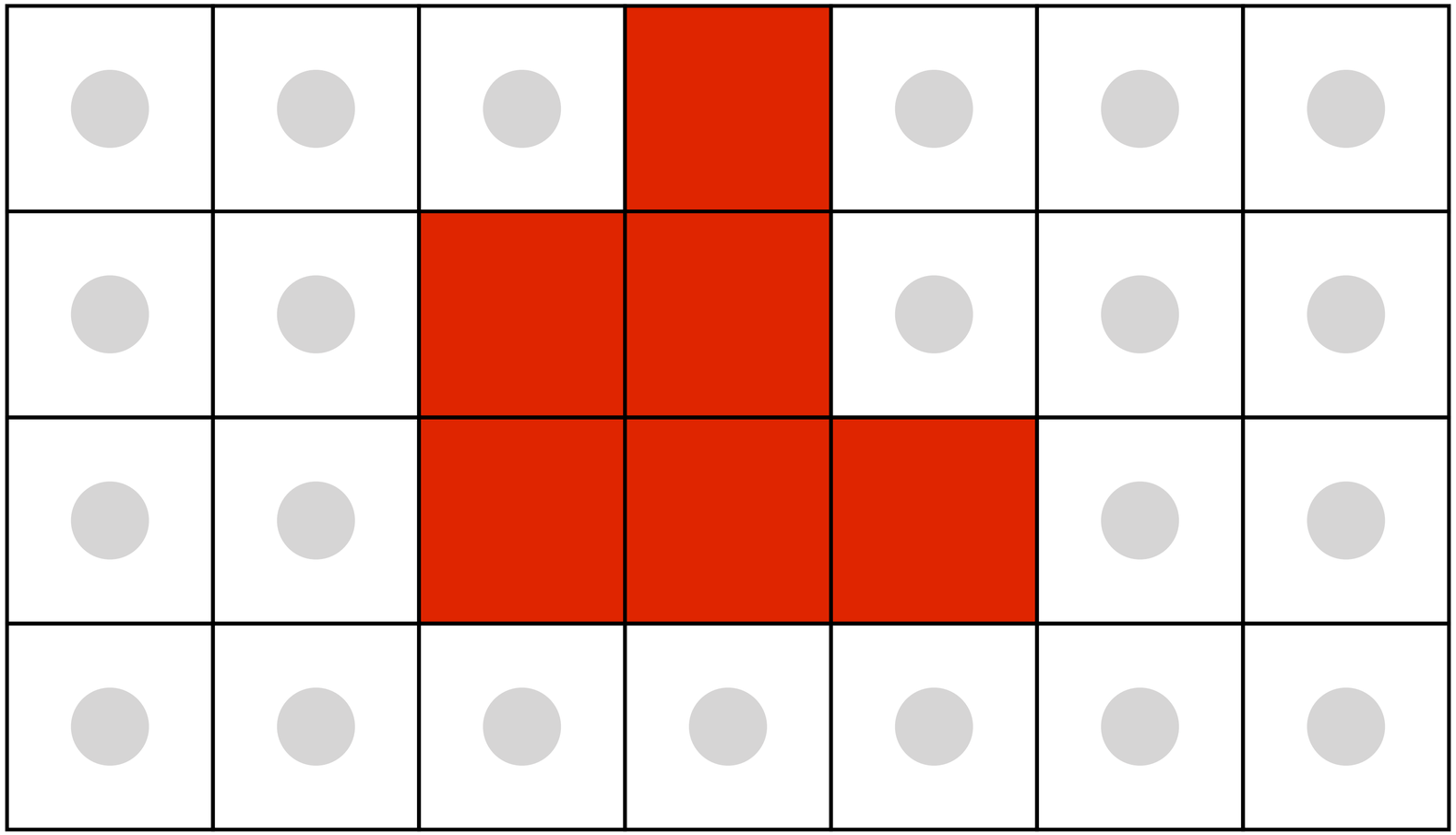
Simple Shading



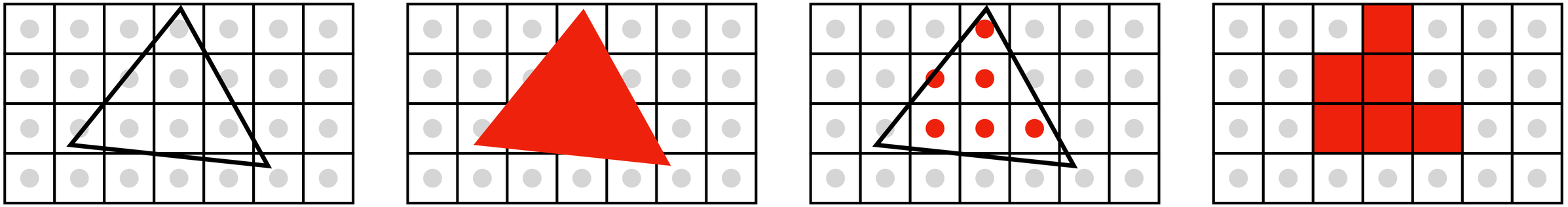
Simple Shading



Simple Shading



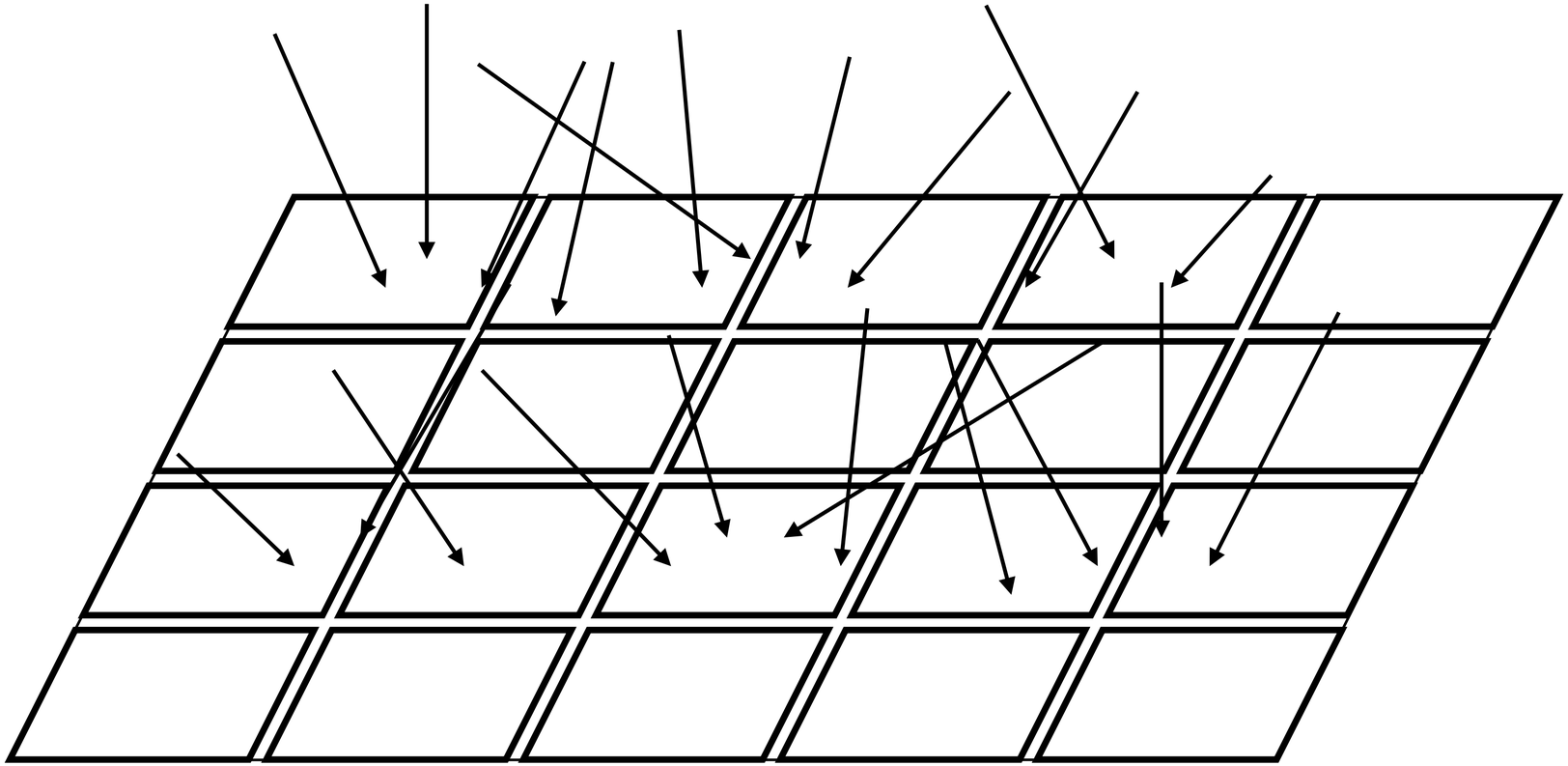
Aliasing in Simple Shading



Remember: each image pixel will be sent to the display, which performs a spatial reconstruction using a box filter. That is, the entire spatial region of a pixel on the display will have the same color.

Effectively, we have sampled a continuous signal (which most likely is not band-limited) at a low frequency (equivalent to image resolution), and then reconstruct the signal using a box filter (on display; not what we can control).

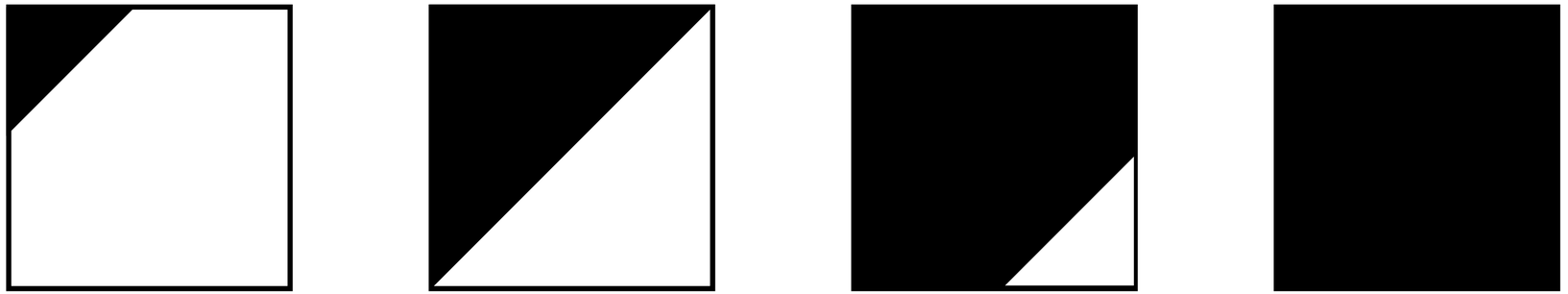
What Do Cameras Do?



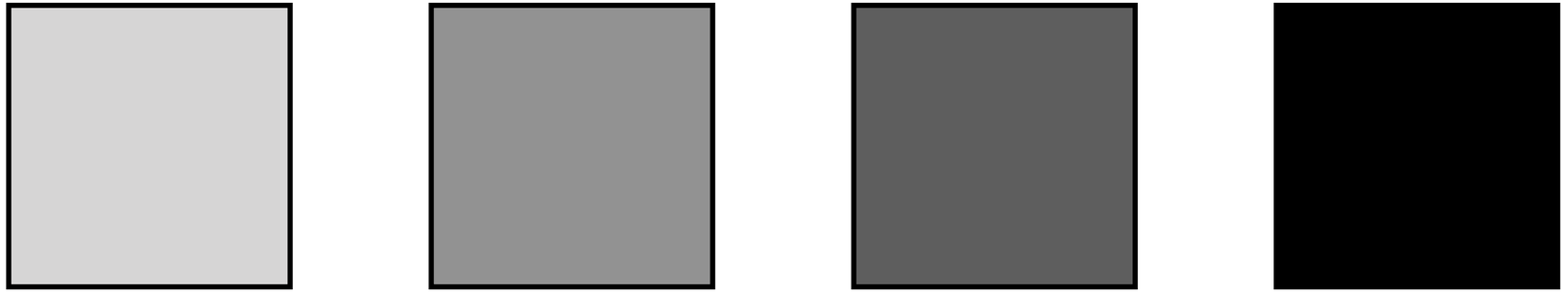
Pixel Array

What Do Cameras Do?

Original



What's Displayed



Super-Sampling

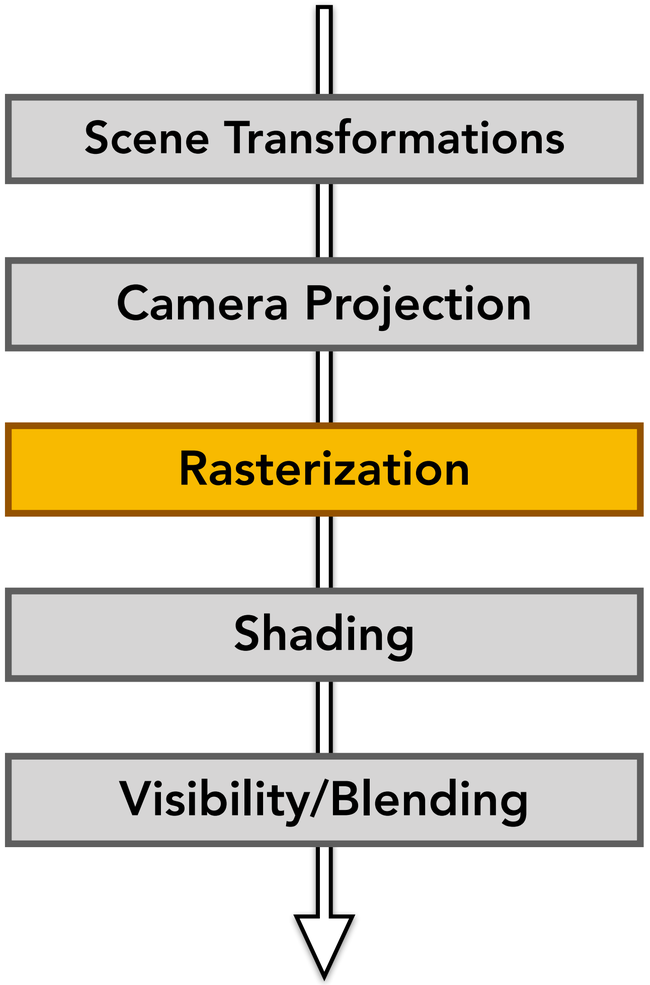
What cameras do is to average energy across the spatial region of a pixel.

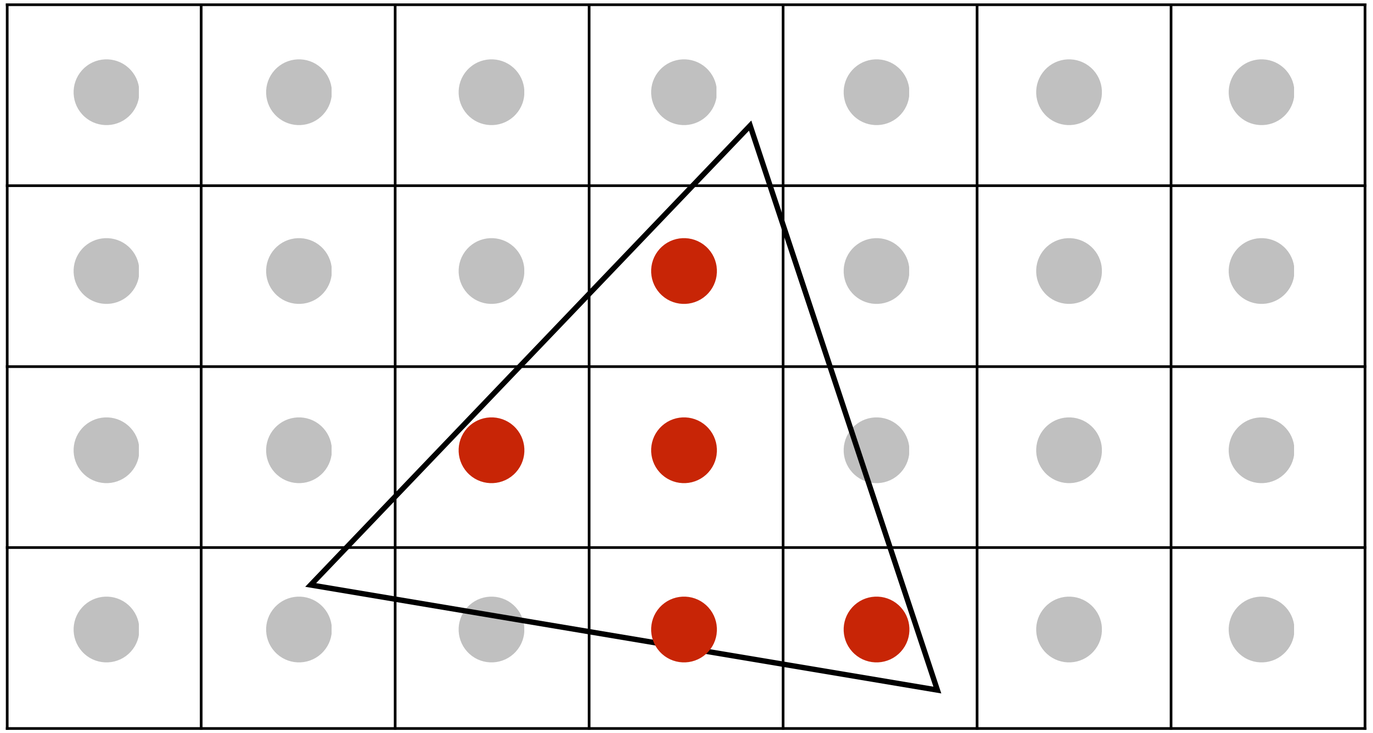
- This is equivalent to applying a box filter and then sample once per pixel.
- The filter size is the same as the physical pixel size, but could also be larger if considering per-pixel micro-lens and anti-aliasing filters.

But in rendering we can’t really take the average, since we don’t know what the continuous function is.

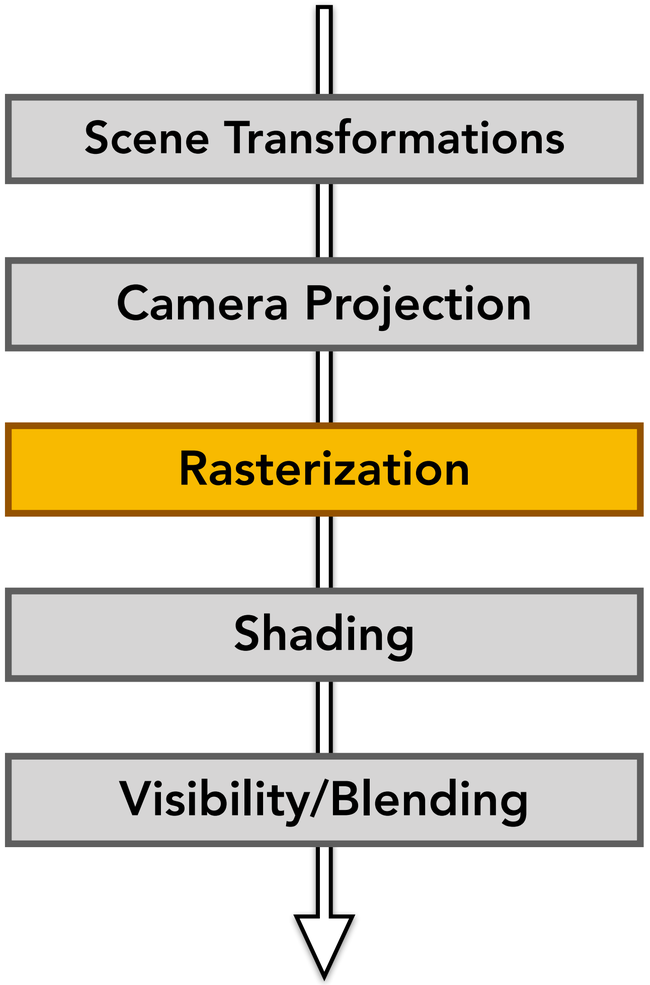
What we do is to approximate this by super-sampling, i.e., sample many times for each pixel, and then average the samples within each pixel.

Point Sampling: One Sample Per Pixel



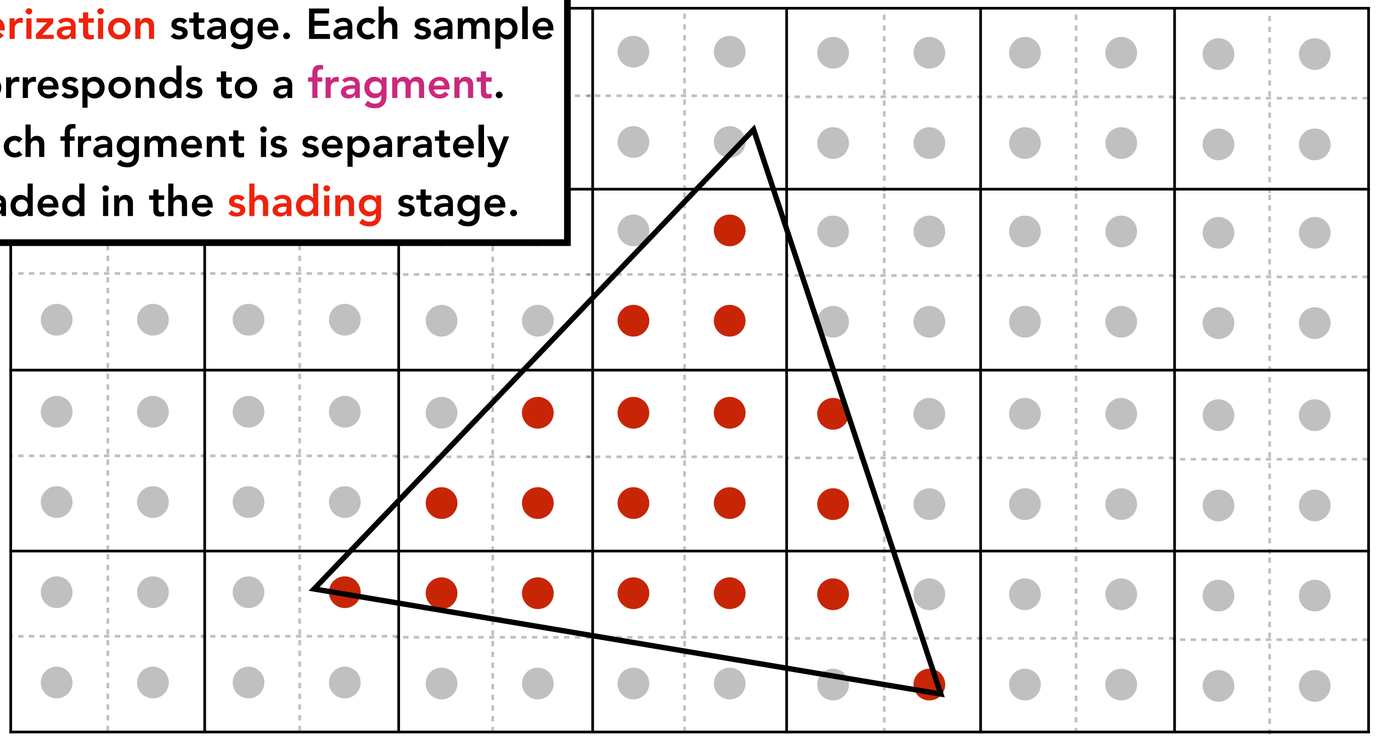


Supersampling: Step 1



Take NxN samples in each pixel.

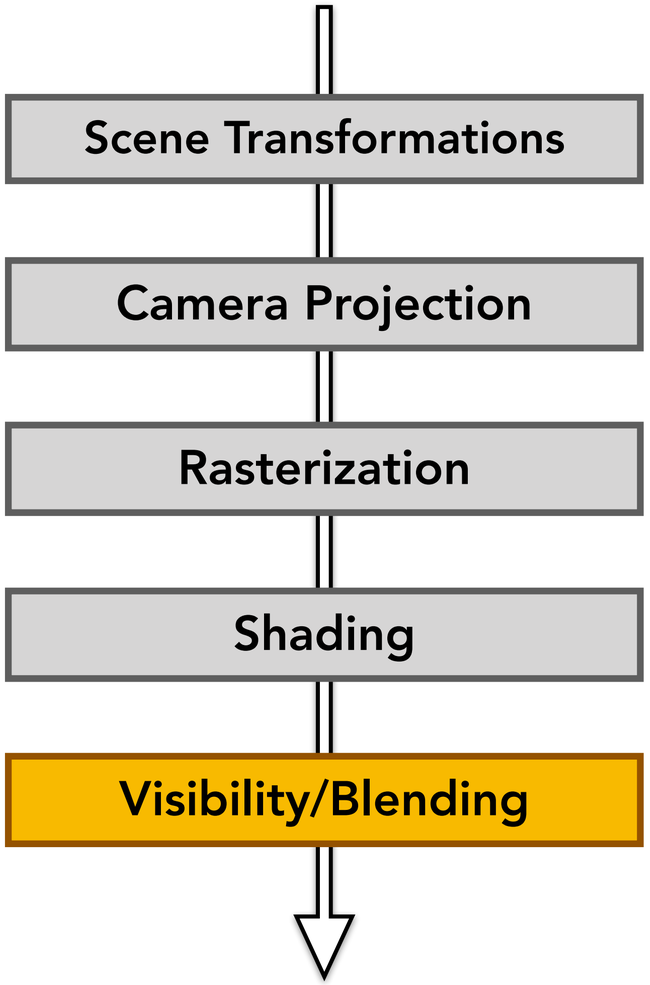
Supersampling is done in the rasterization stage. Each sample corresponds to a fragment. Each fragment is separately shaded in the shading stage.



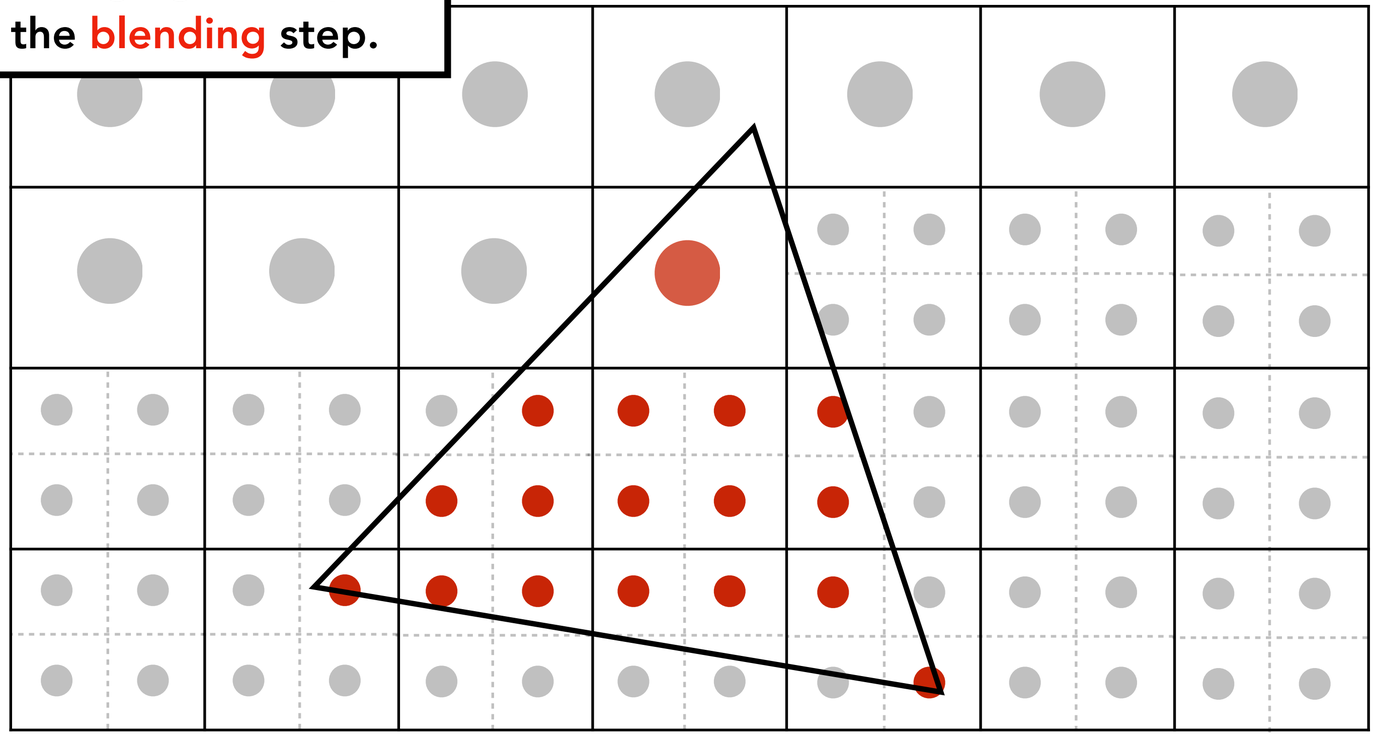
2x2 supersampling

Supersampling: Step 2

Average the NxN samples “inside” each pixel.



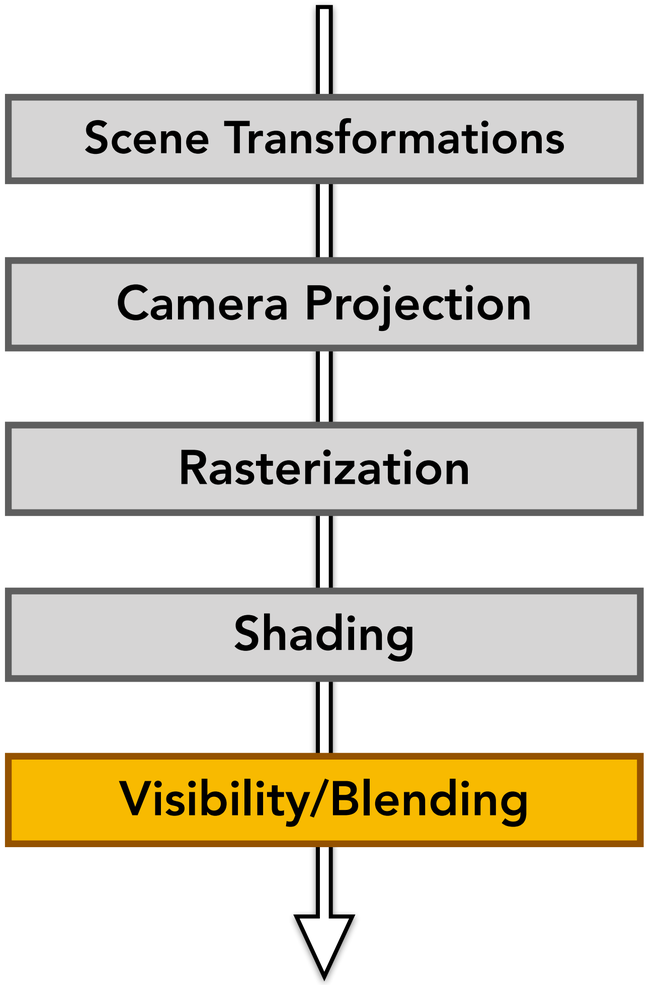
The averaging takes place in the blending step.

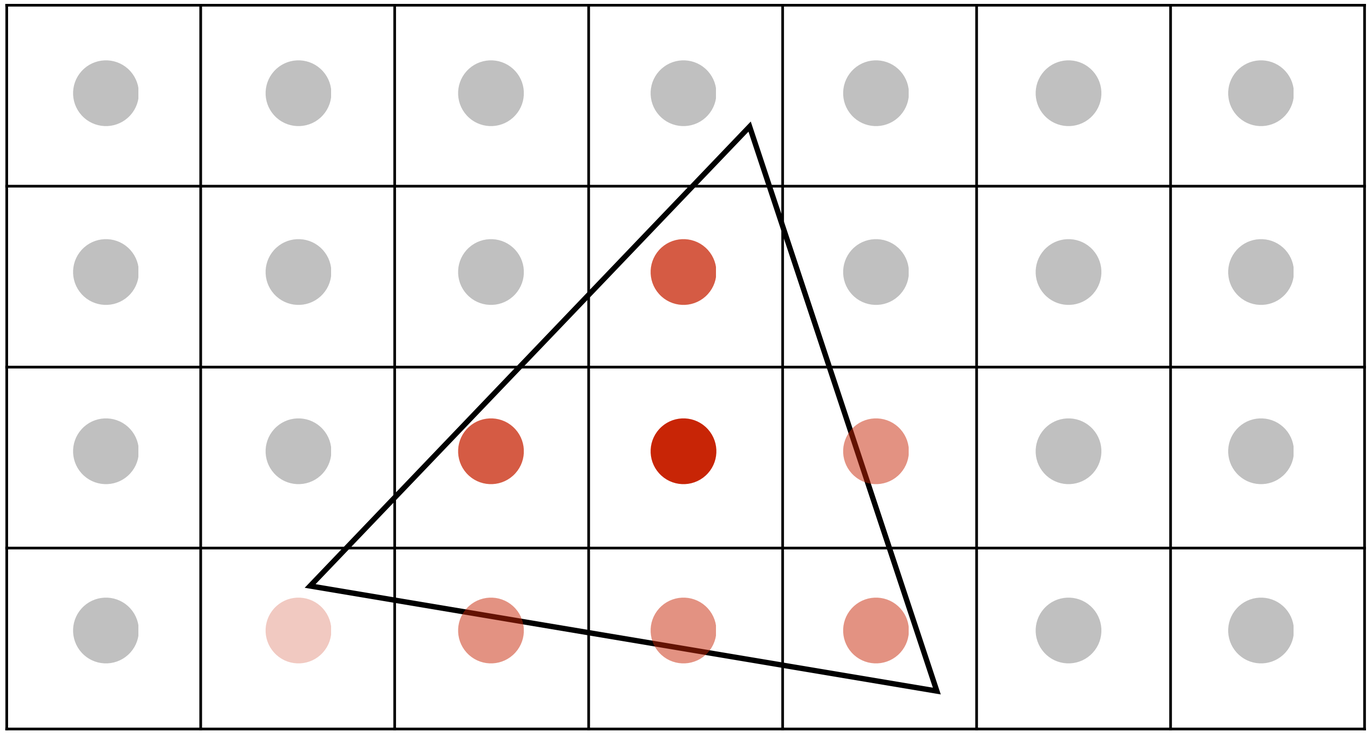


Averaging down

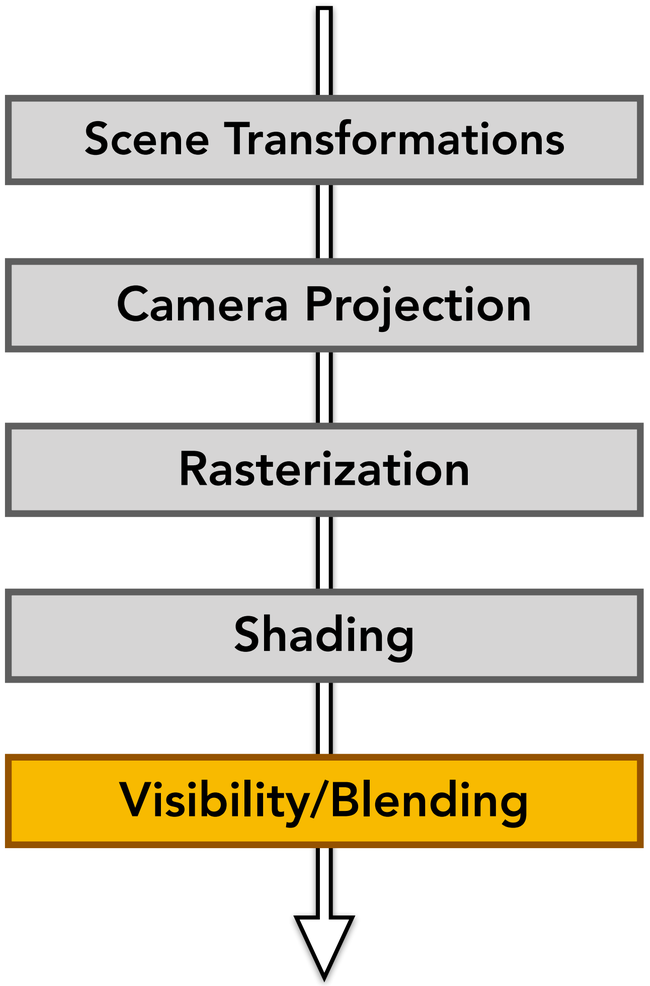
Supersampling: Step 2

Average the NxN samples “inside” each pixel.

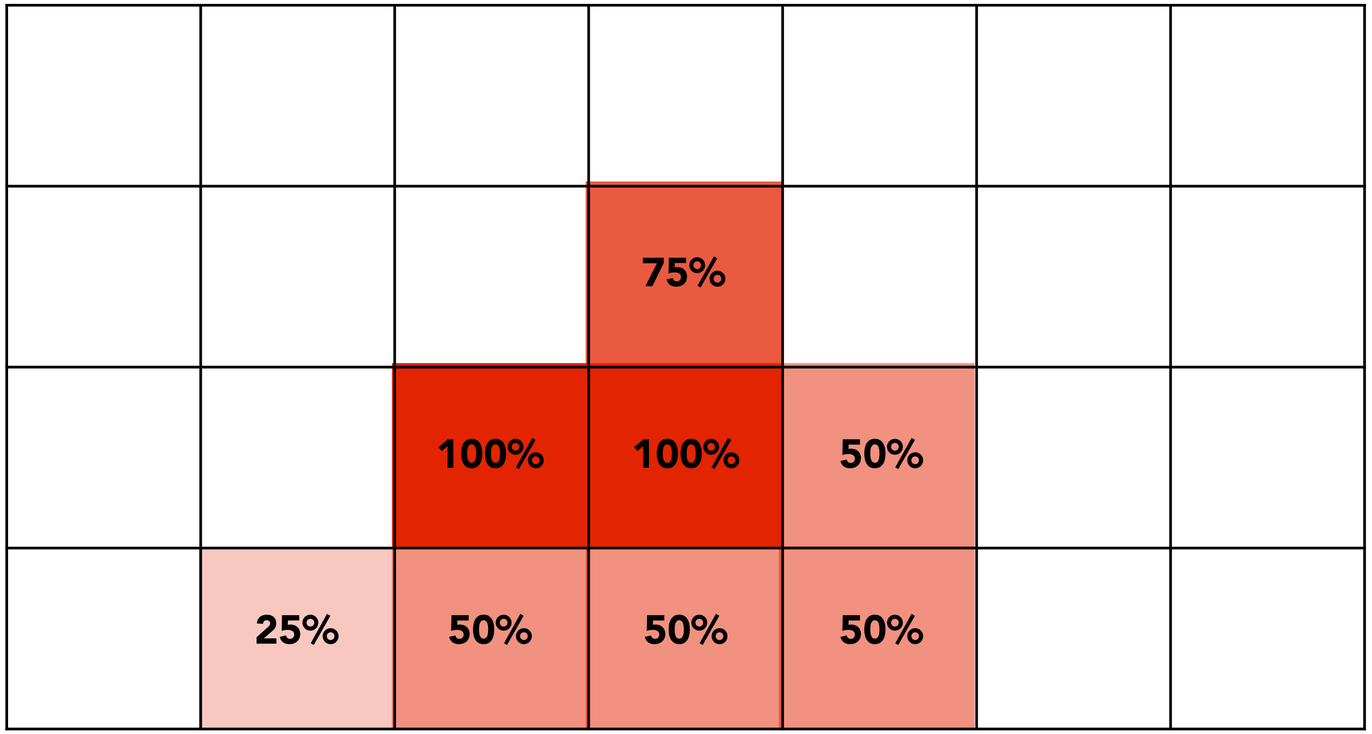




Supersampling: Result



This is the corresponding signal emitted by the display

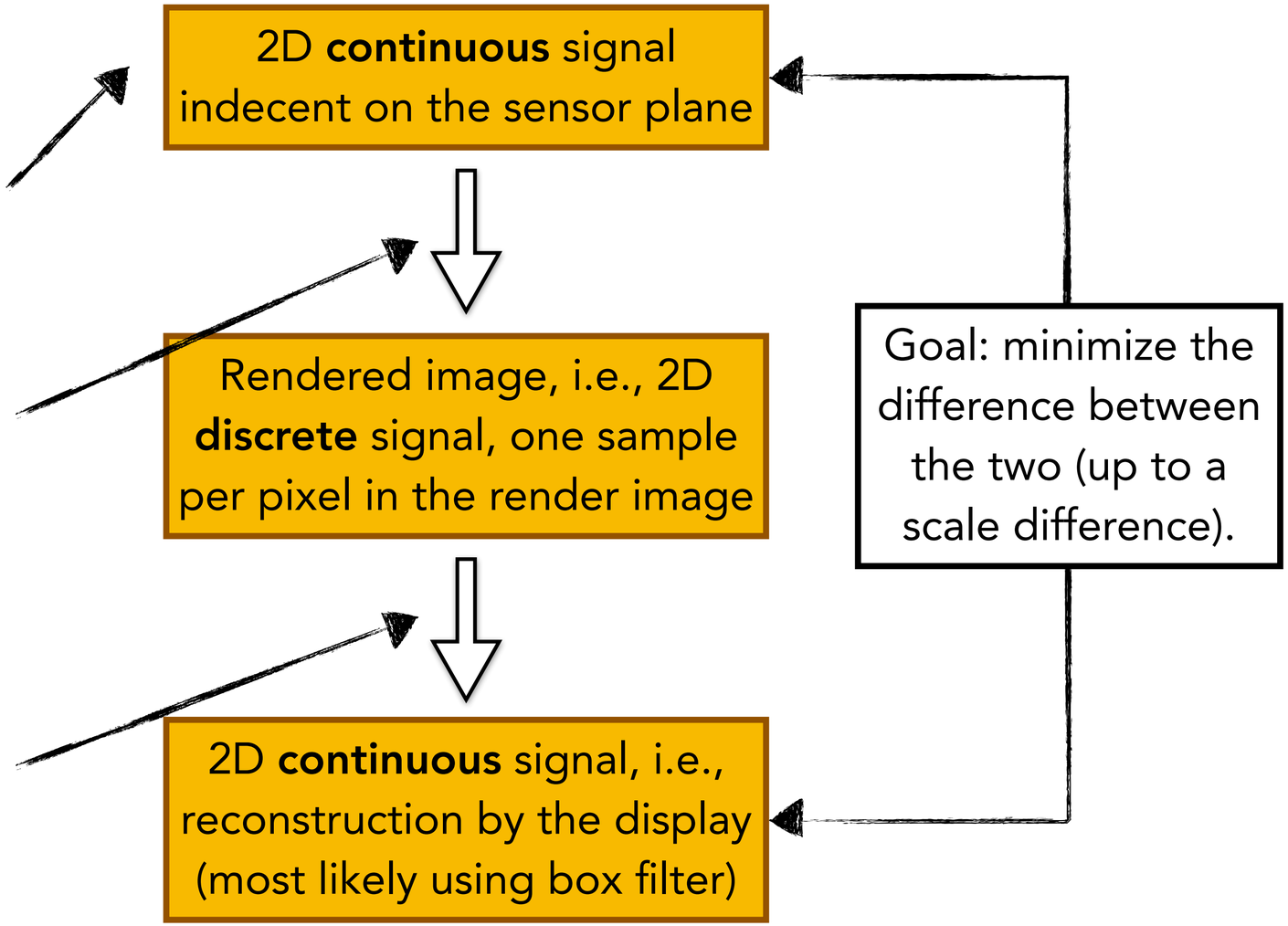


Signal Sampling/Reconstruction Perspective

"Optical image" in camera imaging parlance. Never known analytically. Cameras don't need to know it analytically; pixels simply integrate.

This process is what rendering (or shading specifically) is really about.

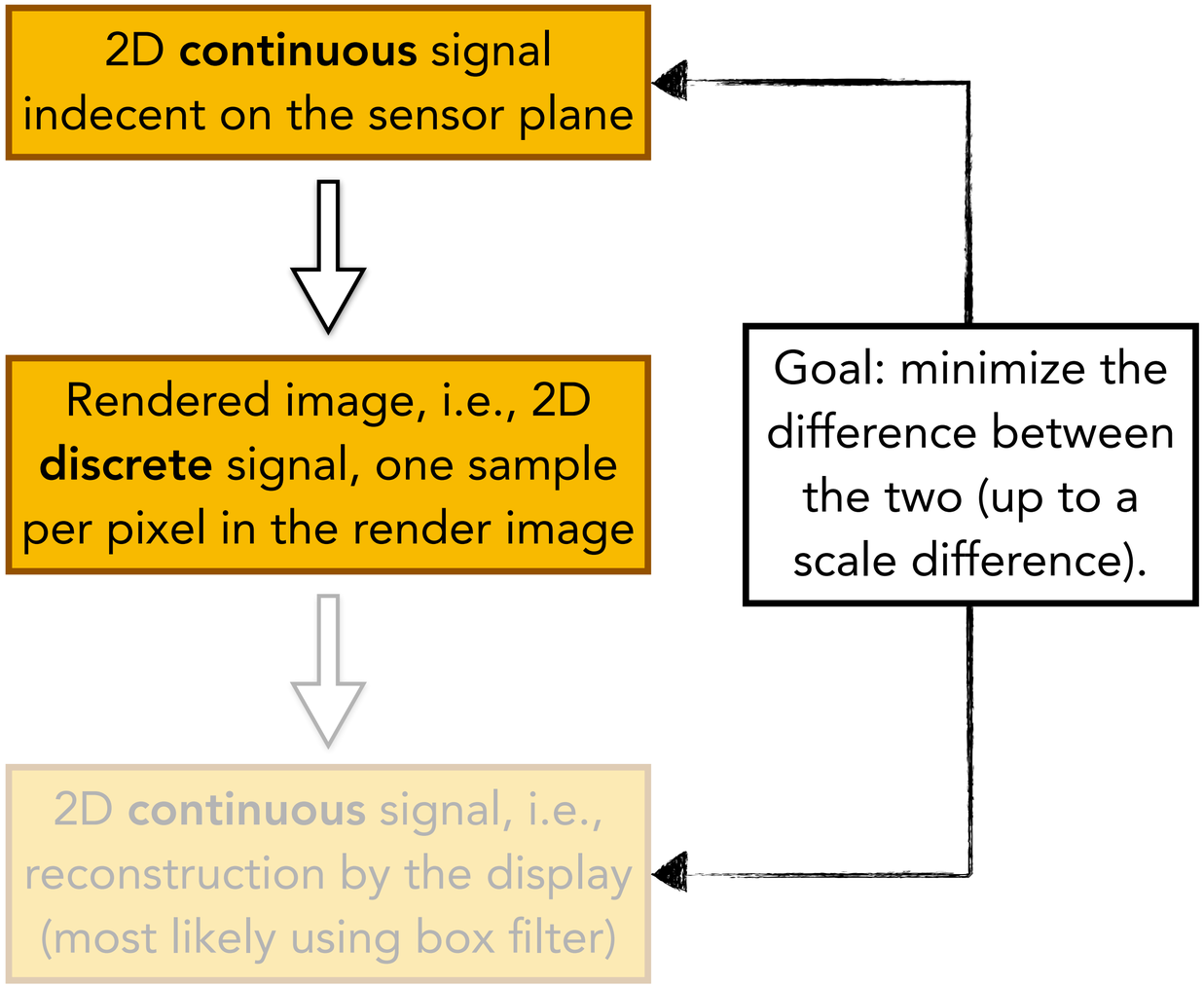
We don't have control over this, but the rendering should ideally take into account this filter. Cameras can't; they always use box filter, but we should!



Ideal Strategy

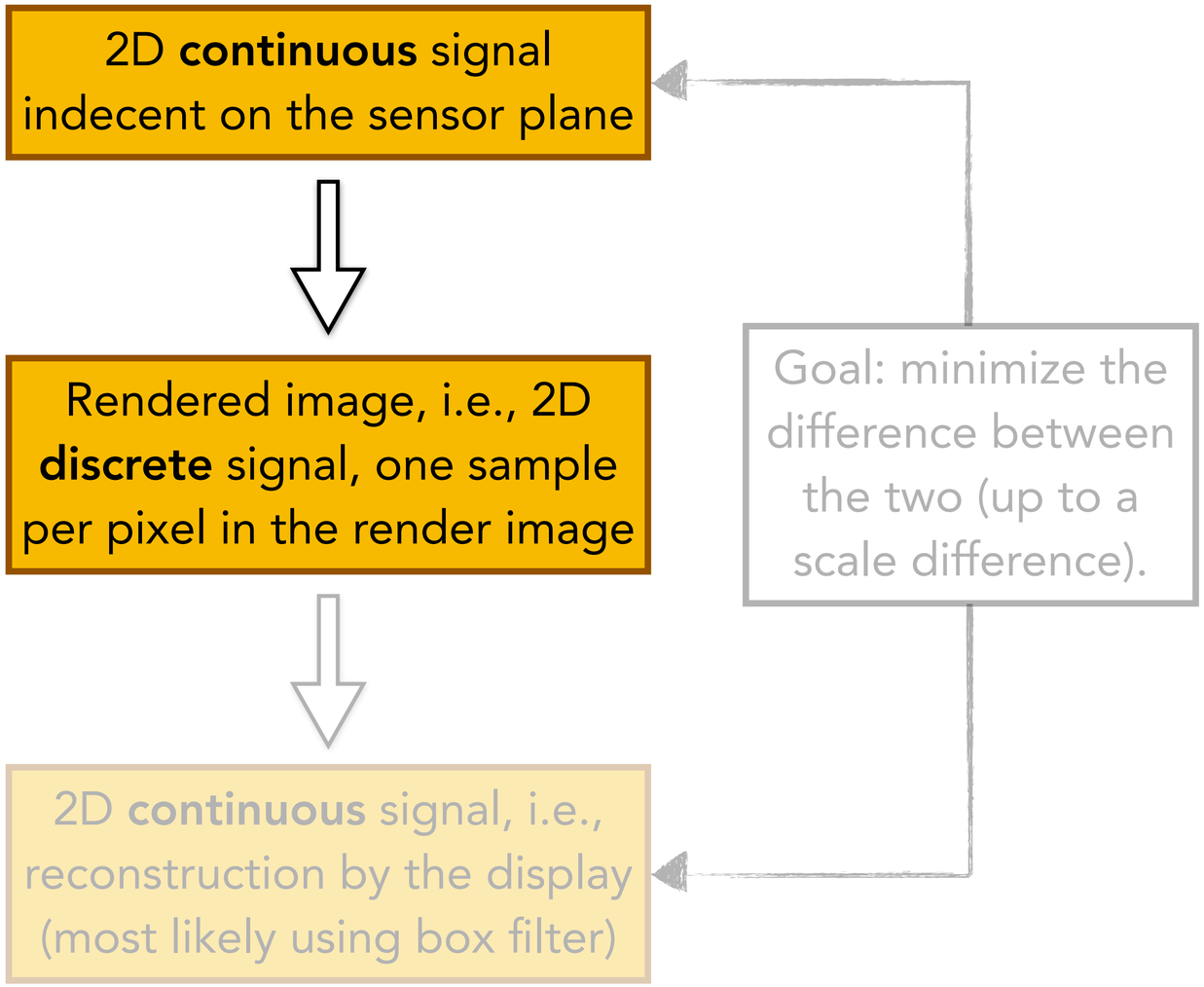
Since the continuous signal most definitely will not be band-limited, any sampling will lead to aliasing.

The idea is to pre-filter the continuous signal to band-limit the signal, since blur is less objectionable than aliasing.

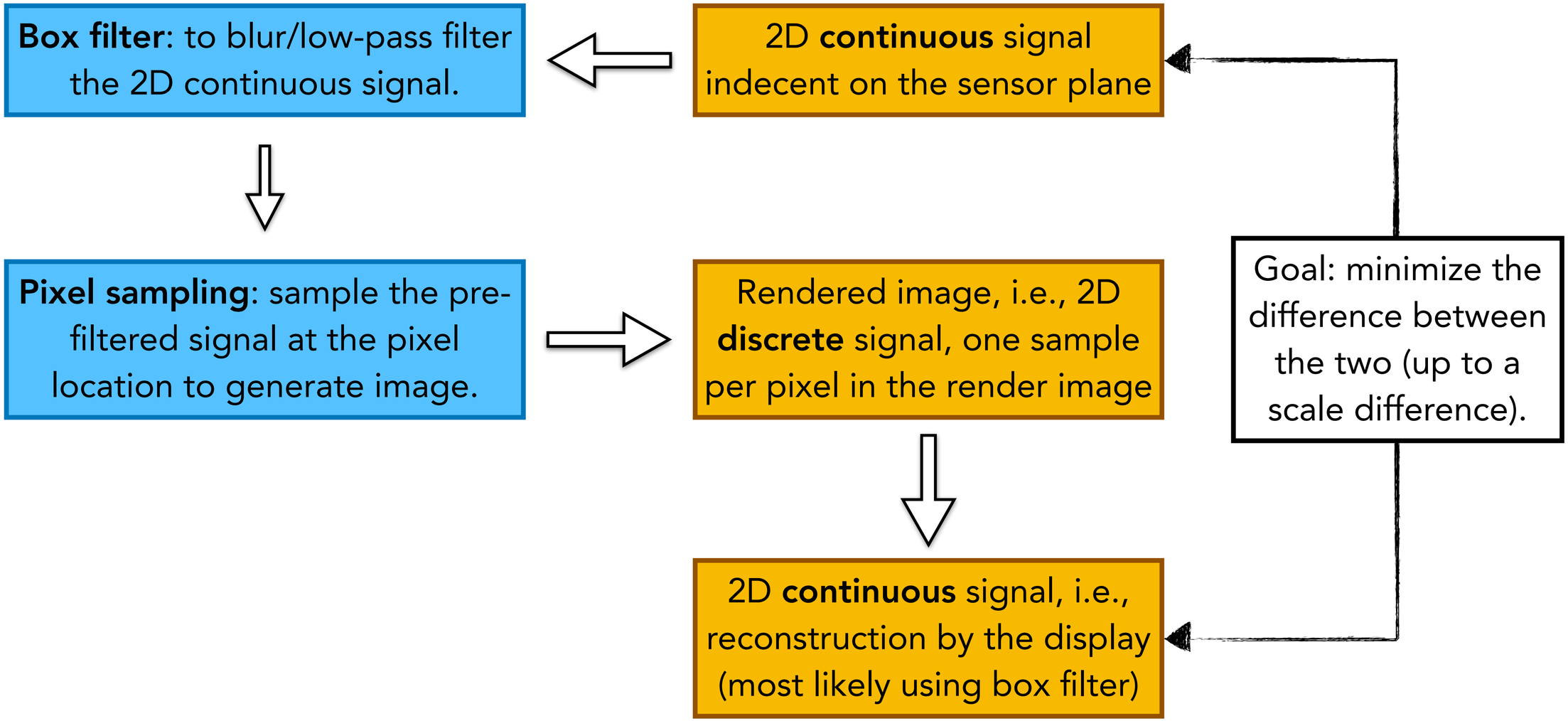


Two Issues with the Ideal Strategy

1. Ideal pre-filtering needs a box function in frequency domain, i.e., a **`sinc`** function in spatial domain
   - but **`sinc`** has infinite support; can't realistically implement it.
2. Usually we don't know the analytical form of the continuous function — cameras do.
   - And they use a box filter at the pixels (with potentially other anti-aliasing filters) for pre-filtering.



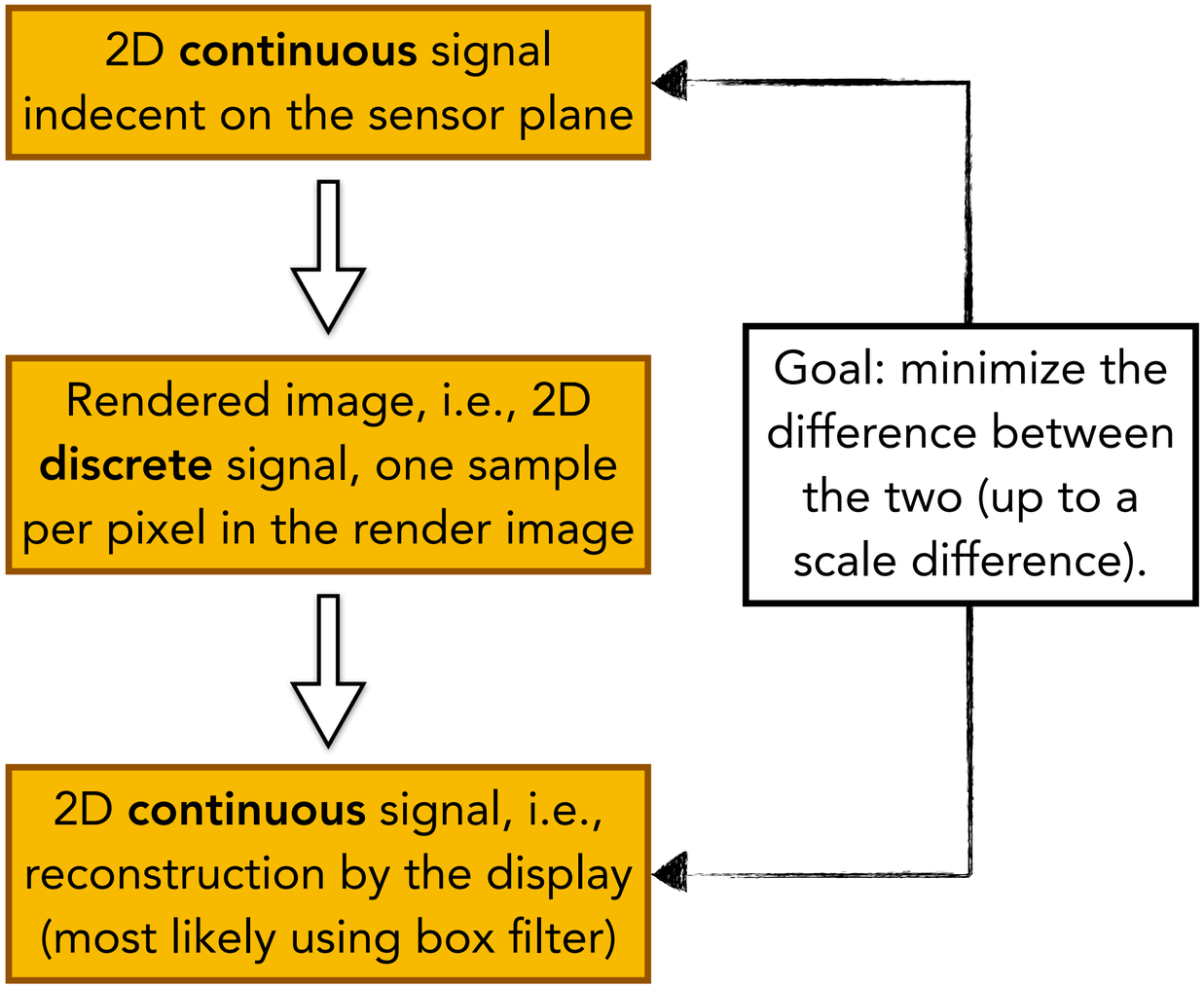
Camera's Strategy



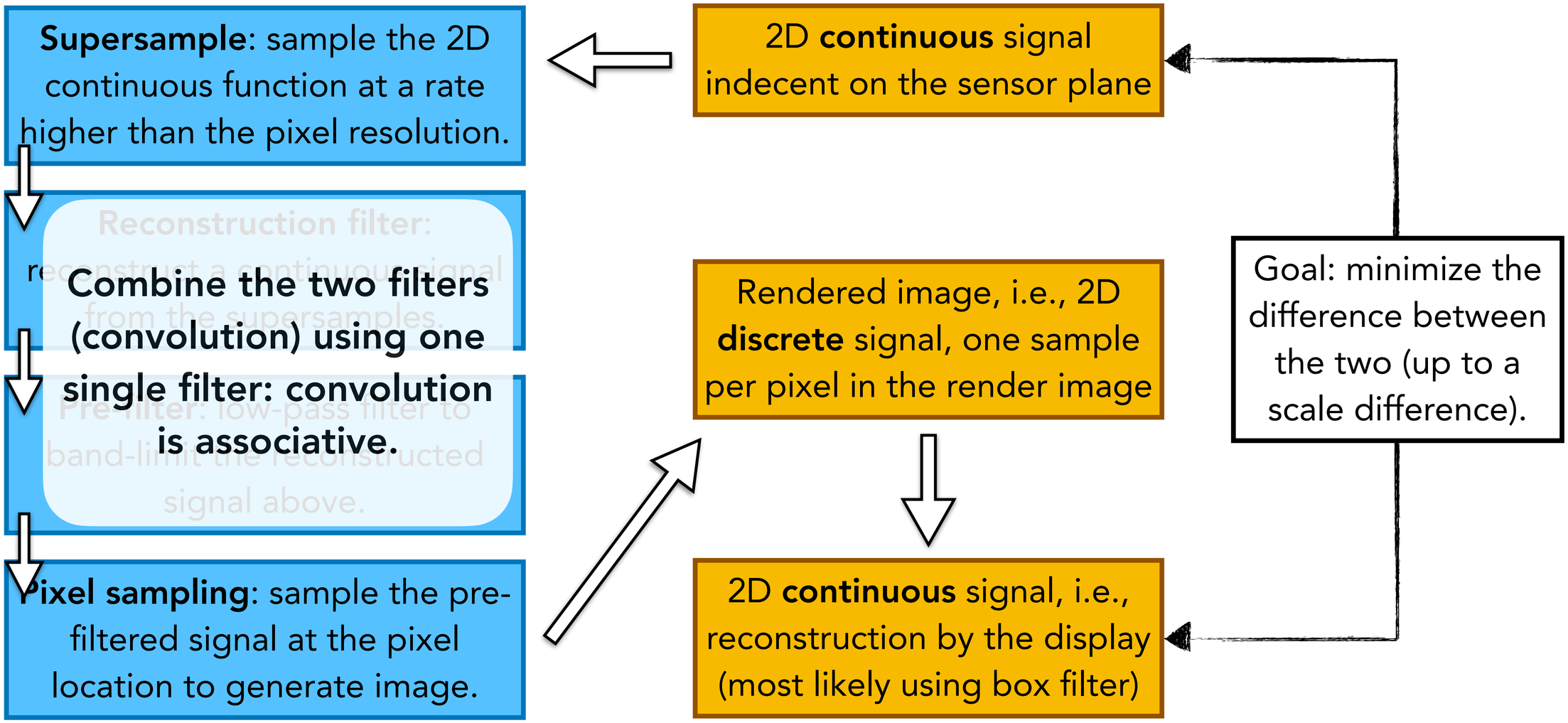
Rendering Strategy

We don’t know the continuous function, so we will sample it and then reconstruct it.

Before the actual pixel sampling, we will take the opportunity to pre-filter the reconstructed continuous signal to band-limit the signal.



Rendering Strategy



A Few Notes

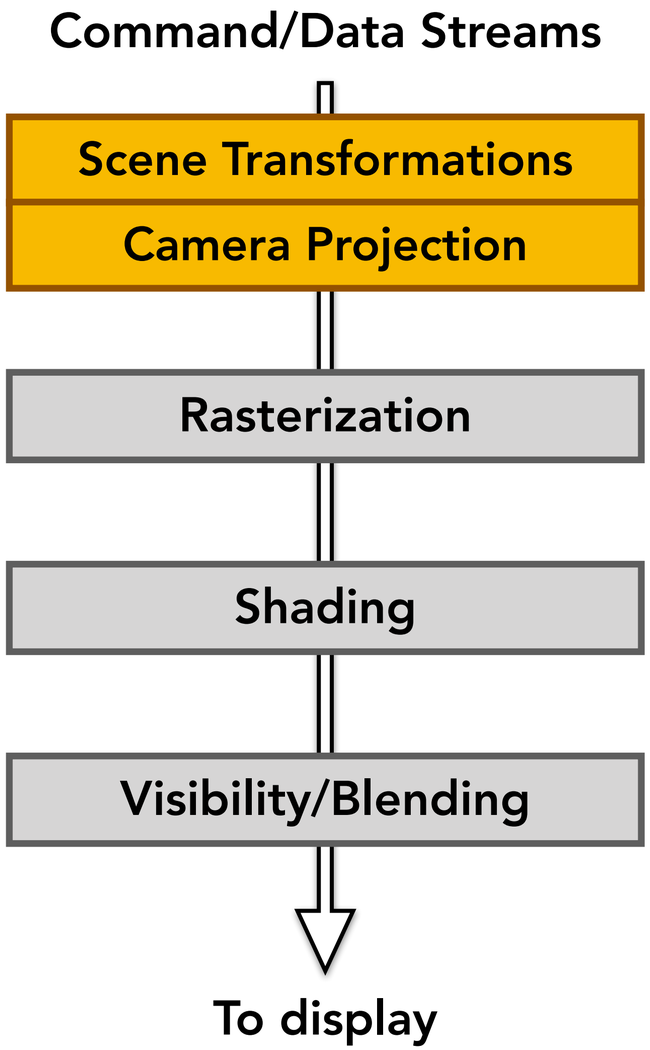
The combined filter can be a box filter, or any other filter. There are many filters that people have experimented; ultimately, there is virtually no hope for perfect reconstruction on the display, so it's all about the empirical rendering quality.

- See: https://pbr-book.org/3ed-2018/Sampling_and_Reconstruction/Image_Reconstruction

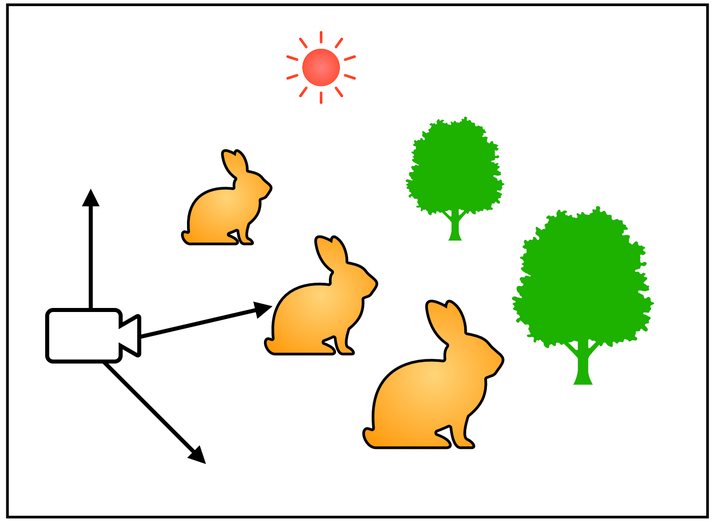
Can also use non-uniform sampling, or filter beyond a pixel's spatial region.

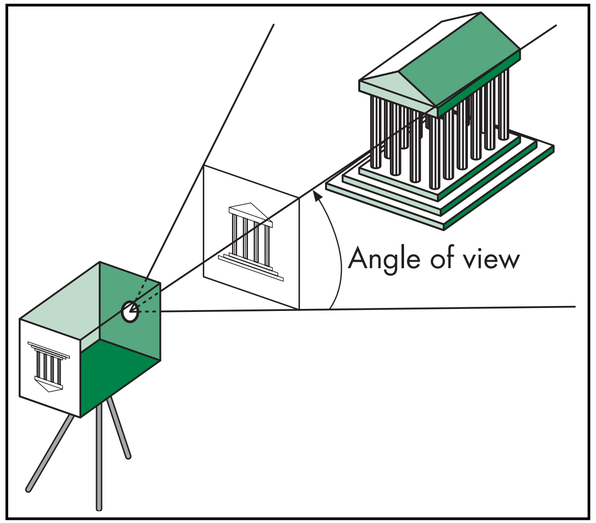
This discussion is general to any shading, not just in rasterization pipeline.

Rasterization Pipeline Summary

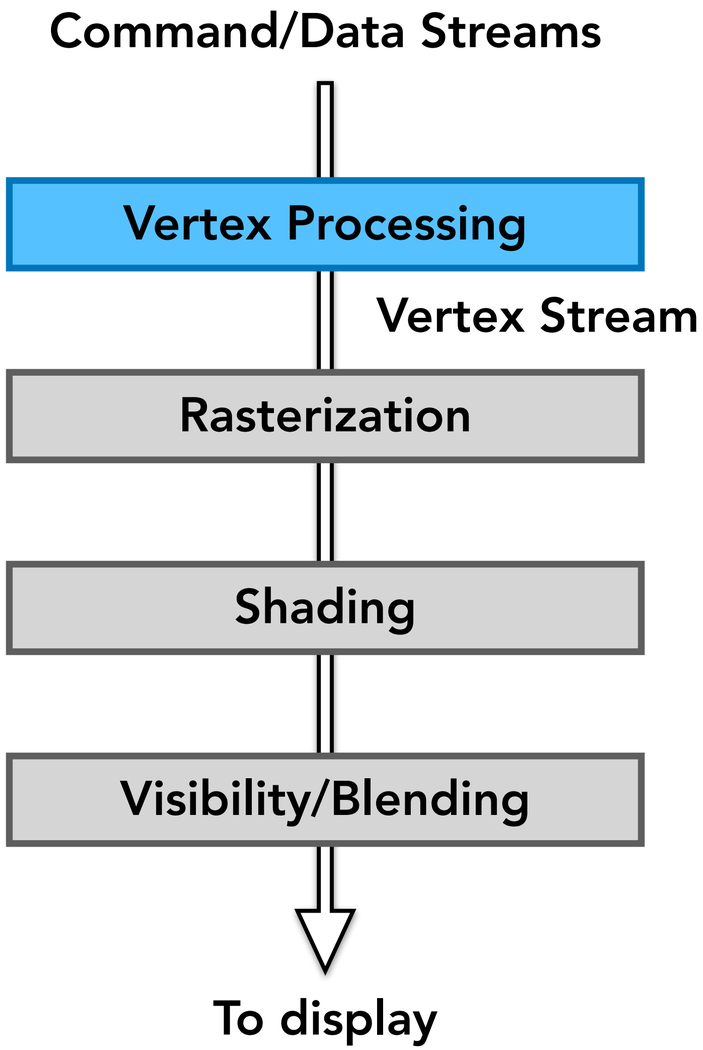


Both manipulate triangle vertices and so are lumped together as "**vertex processing**", which is made **programmable** in rasterization pipeline to allow custom transformations.





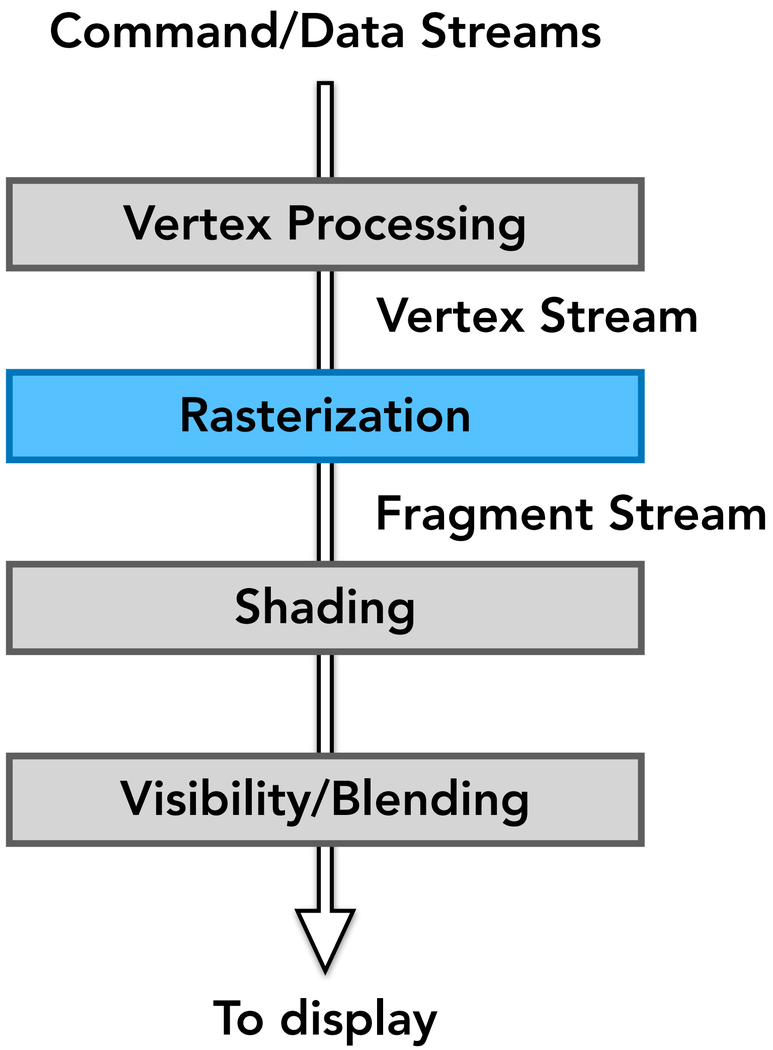
Rasterization Pipeline Summary



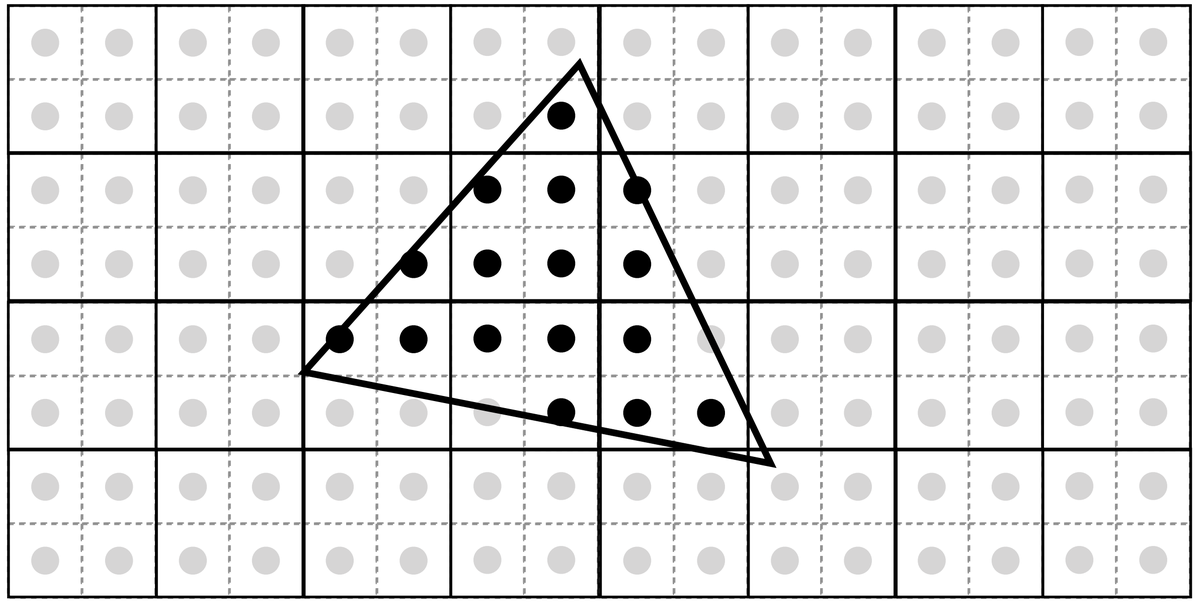
A **vertex shader** that describes how to transform a vertex; the shader is applied to all vertices.

```
uniform float t;
attribute vec4 vel;

const vec4 g = vec4(0.0, -9.8, 0.0);

void main() {
  vec4 position = gl_Vertex;
  position += t*vel + t*t*g;

  gl_Position = gl_ModelViewProjectionMatrix * position
}
```

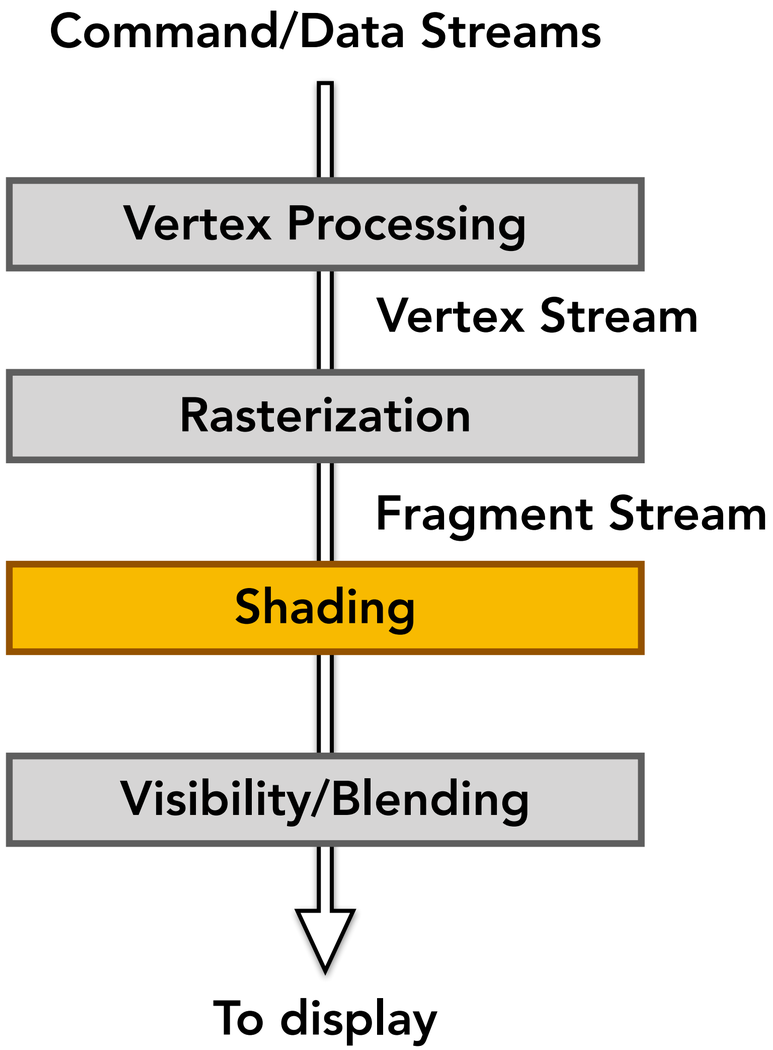
Rasterization Pipeline Summary



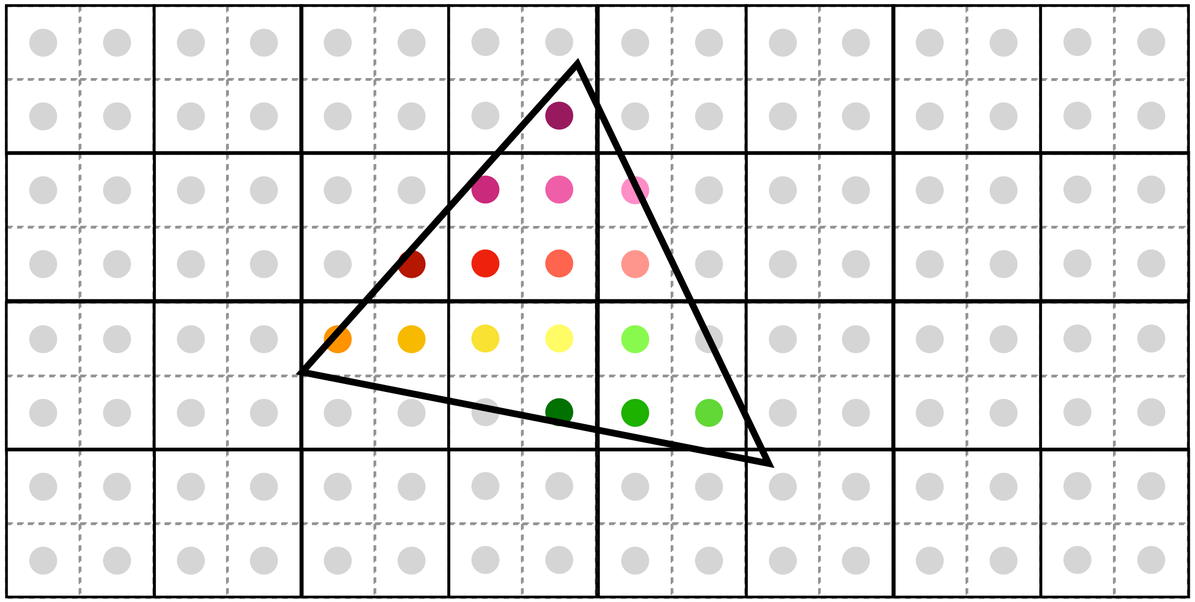
Potentially super-sampling



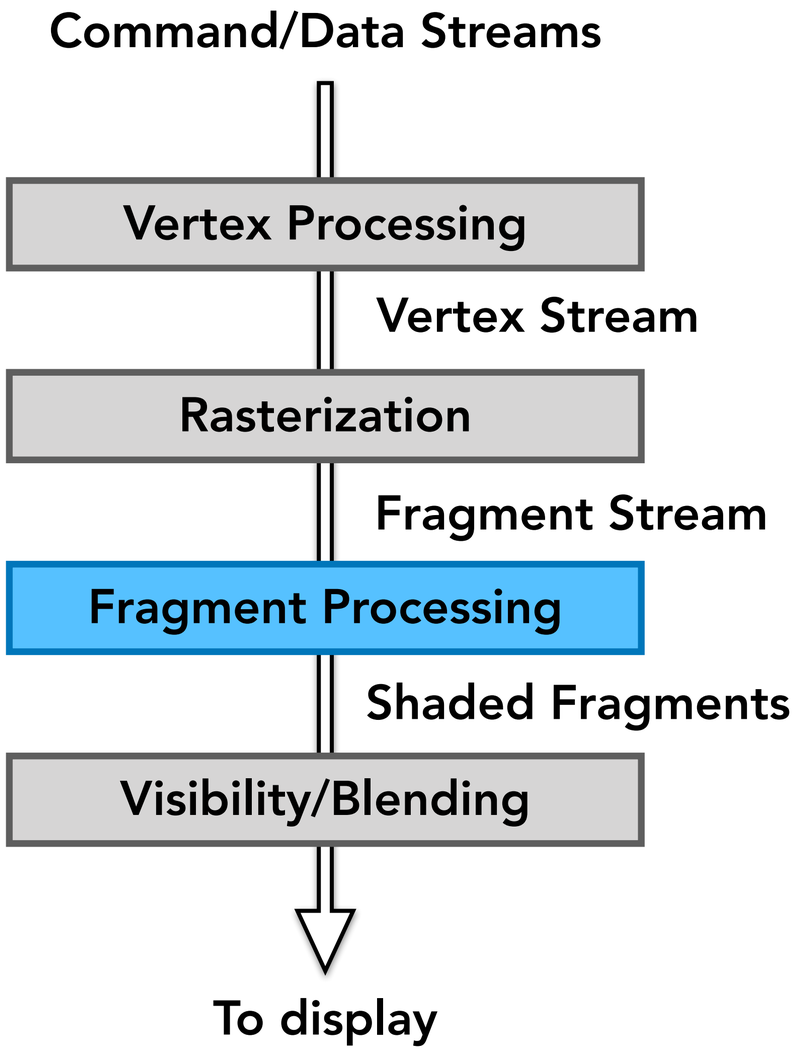
Rasterization Pipeline Summary



Calculating colors for each fragment. This is abstracted as "**fragment processing**", which, like vertex processing, is programmable in rasterization pipeline.



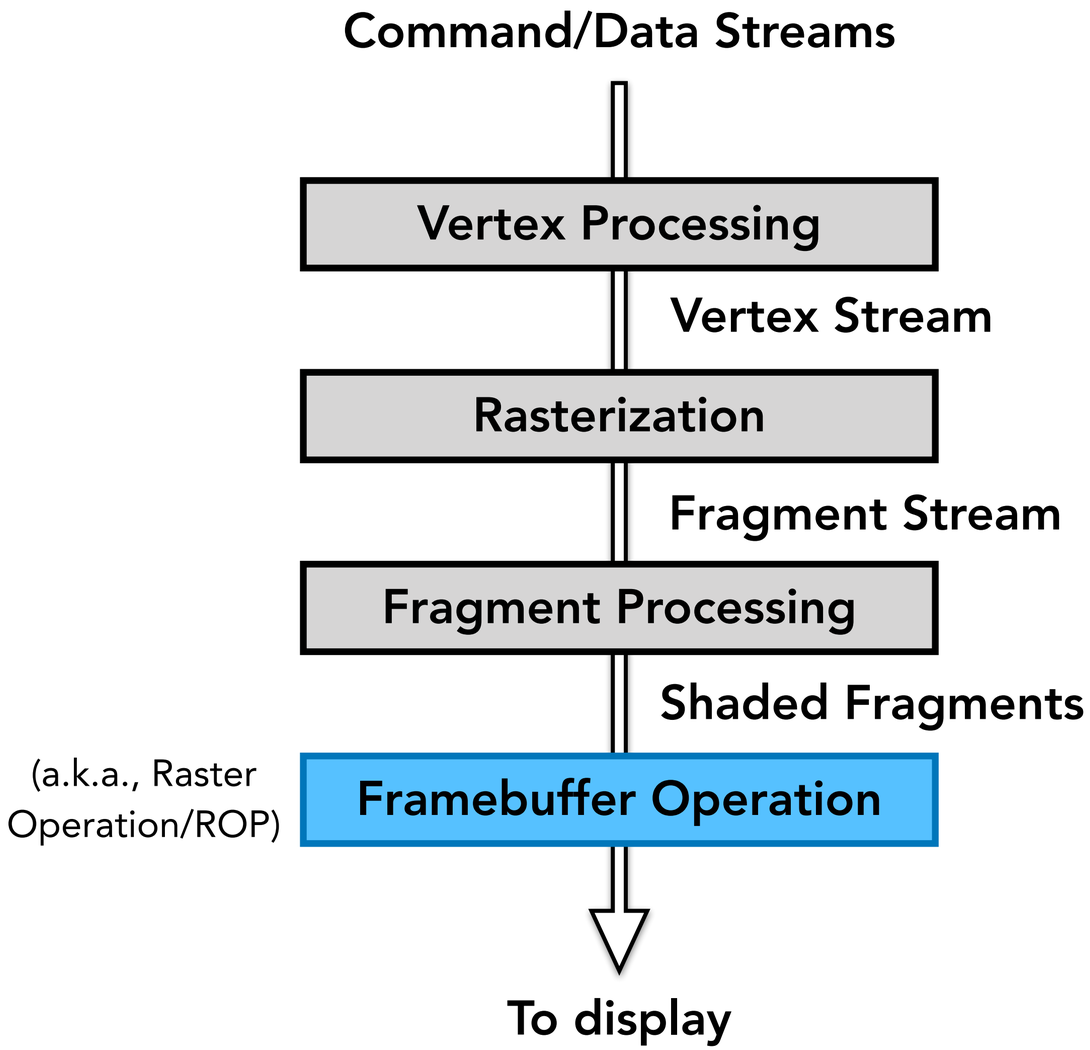
Rasterization Pipeline Summary



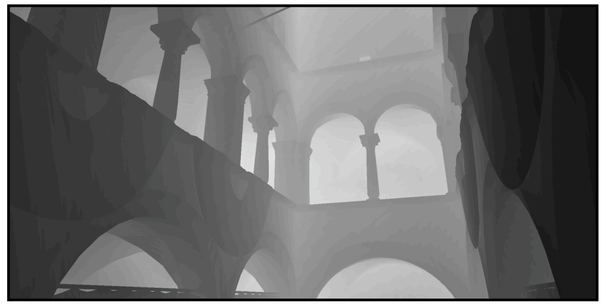
A **fragment shader** code that calculates fragment color; the shader is applied to all fragments. Texture mapping is fragment processing too (later).

```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader() {
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);

  gl_FragColor = vec4(kd, 1.0);
}
```

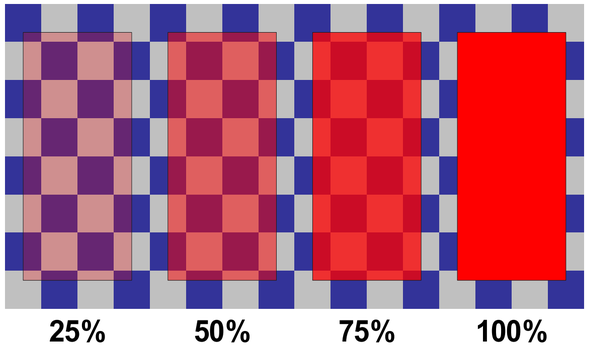
Rasterization Pipeline Summary



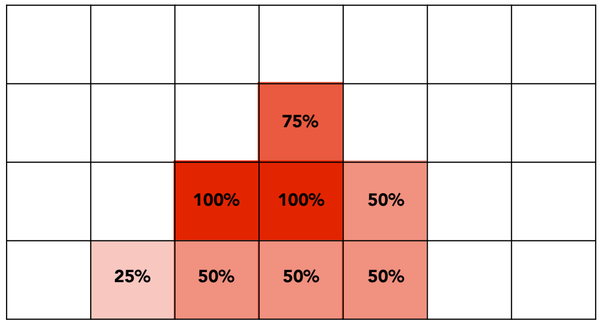
Z-buffer visibility test



Blending

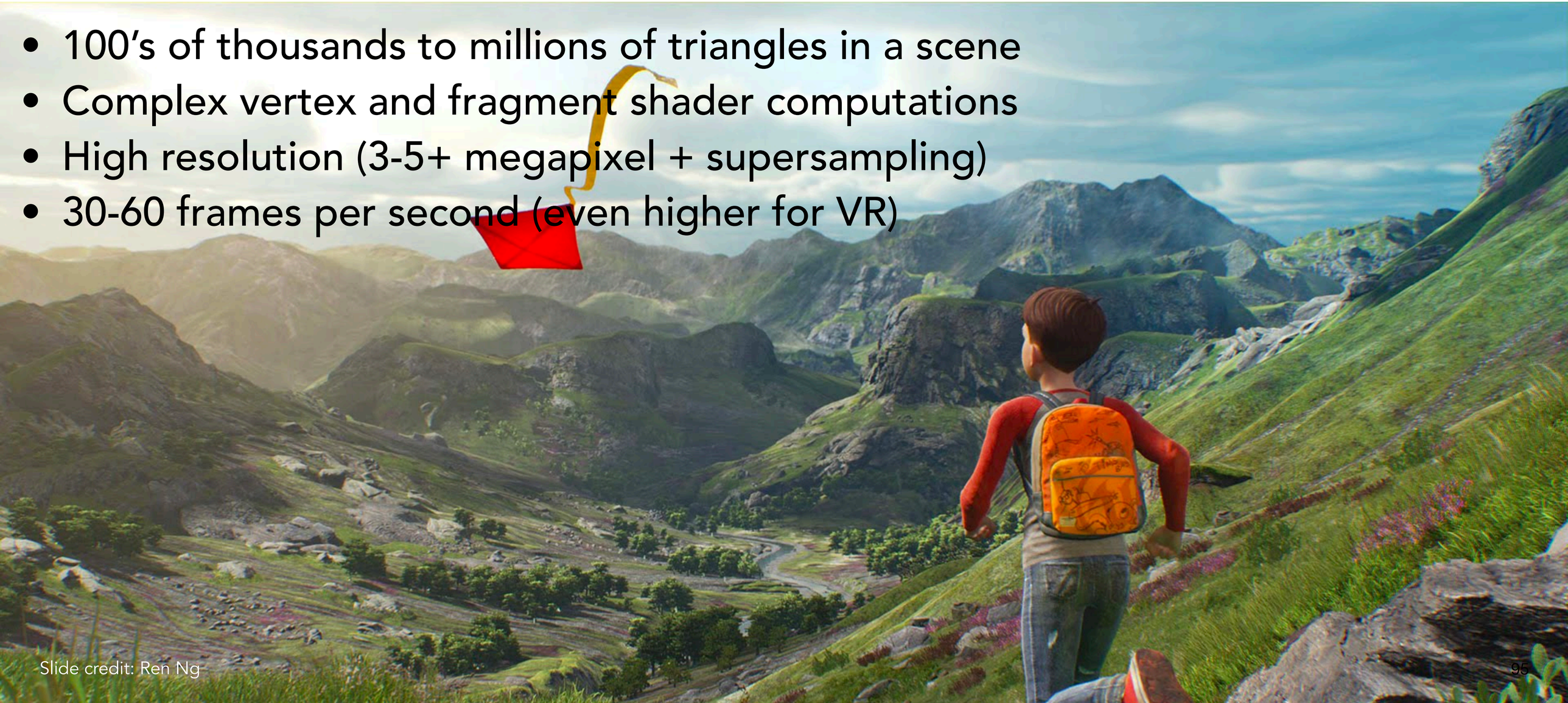


Anti-aliasing



Massively Parallel Processing

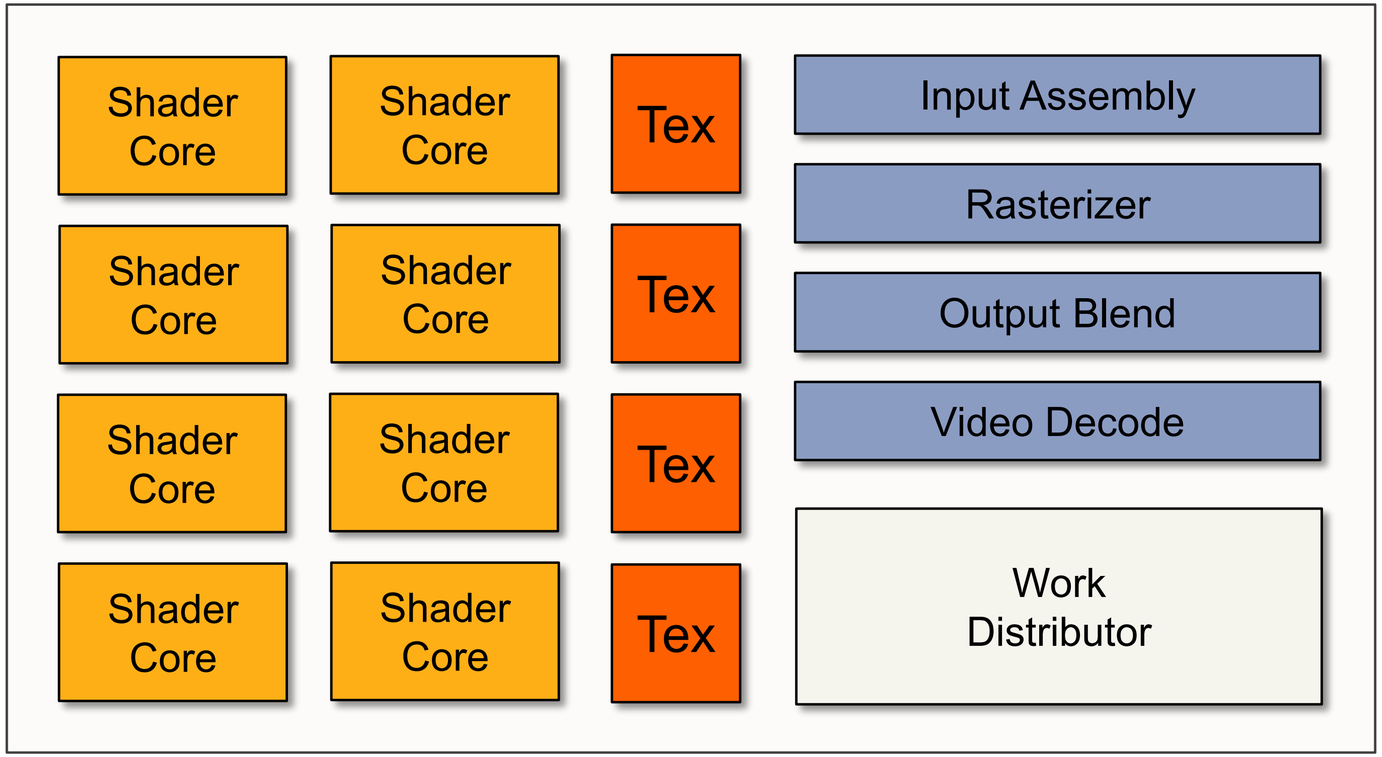
- 100’s of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution (3-5+ megapixel + supersampling)
- 30-60 frames per second (even higher for VR)



Slide credit: Ren Ng

Rasterization GPU Hardware

What's in a GPU?



Heterogeneous chip multi-processor (highly tuned for graphics)

GPU Hardware (Maxwell)

Fixed-function units
(no programmability)

- Each Stream Multiprocessor (SM) contains massively parallel computing units for processing vertices and fragments.
- There are many SMs. Each can execute a different program.

https://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/3

Inside an SM

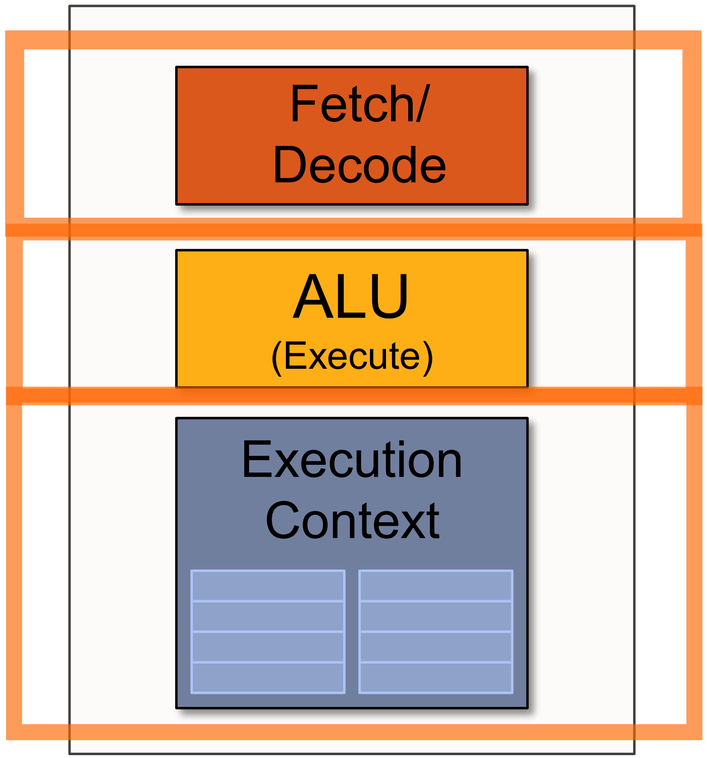
Running shaders
(vertex and fragment)

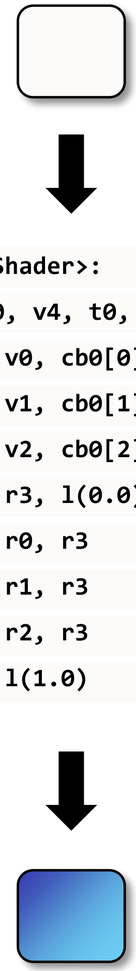
Cache for texture maps
(exploits data access
patterns to the texture map)

Texture sampling units

https://www.anandtech.com/show/8526/nvidia-geforce-gtx-980-review/3

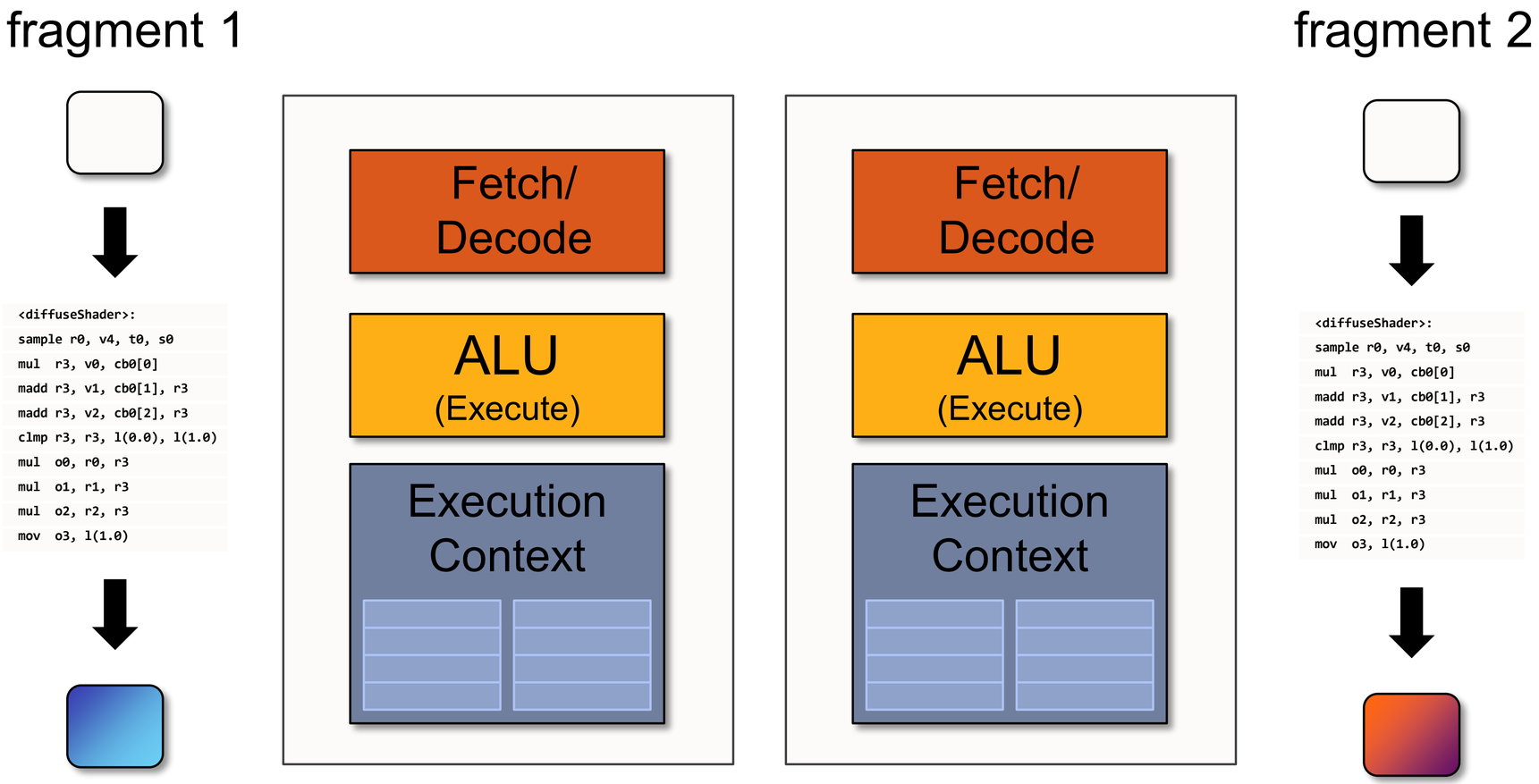
Execute shader



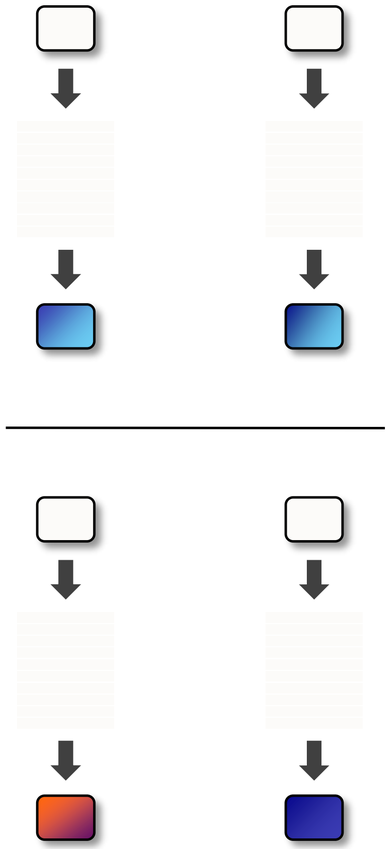


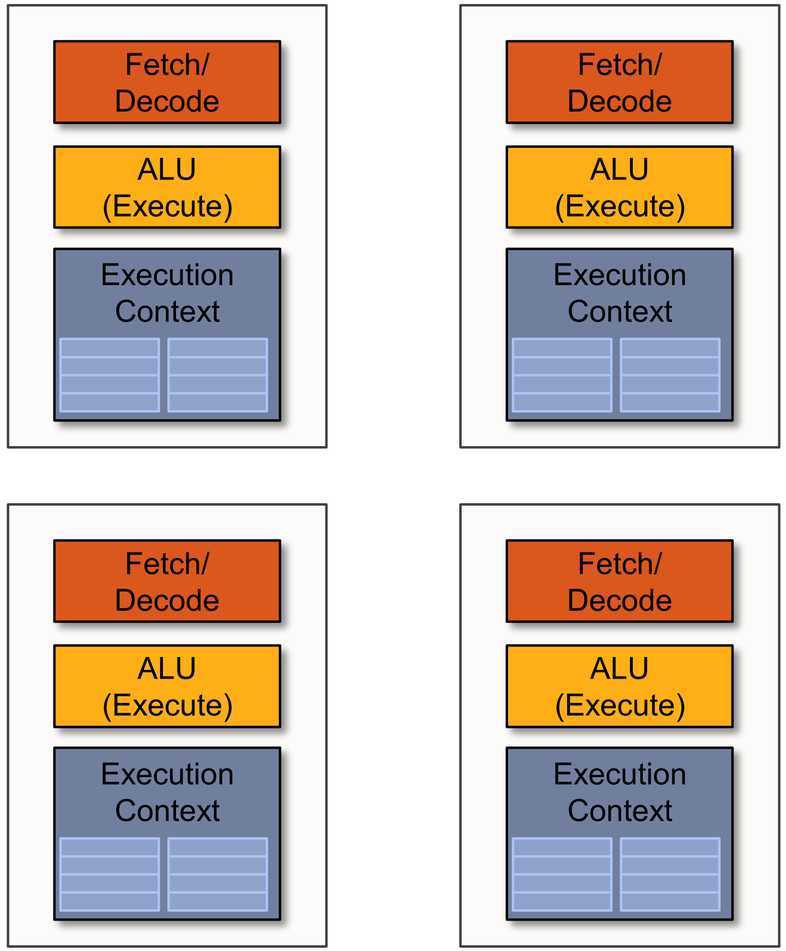
```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

Two cores (two fragments in parallel)



Four cores (four fragments in parallel)

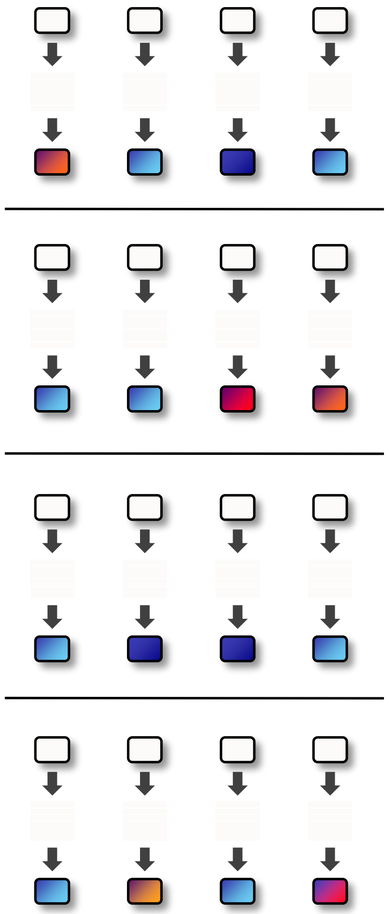




Sixteen cores (sixteen fragments in parallel)

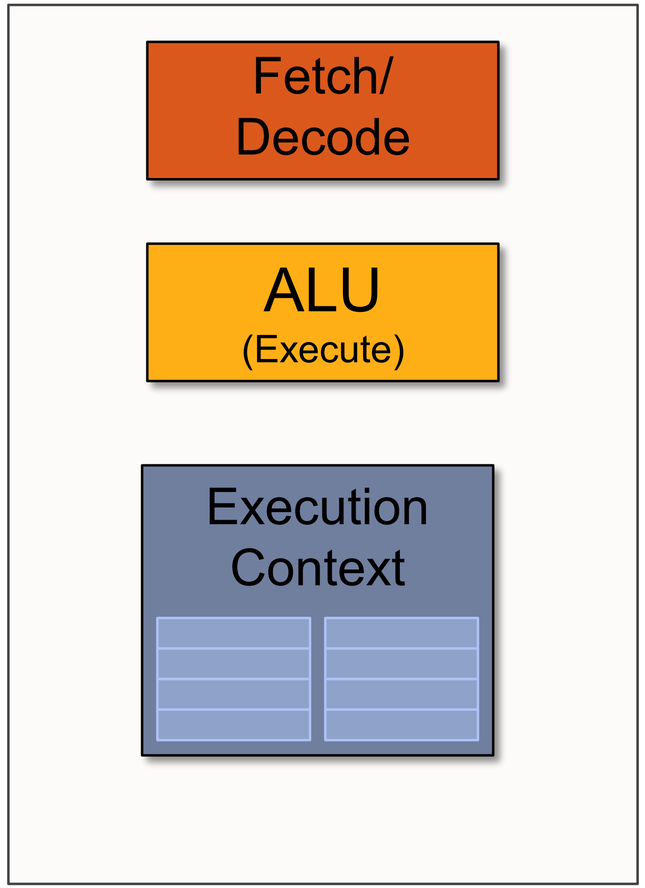
16 cores = 16 simultaneous instruction streams

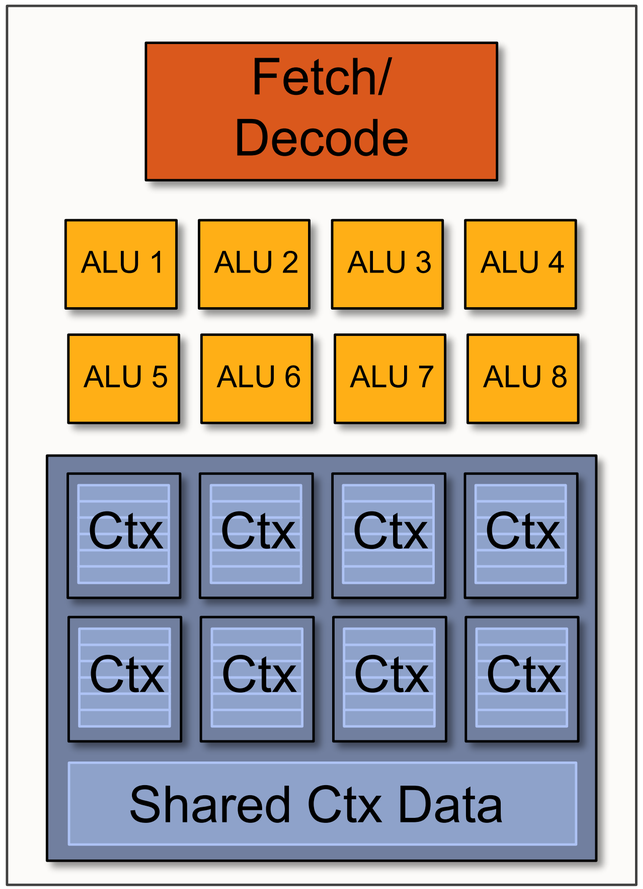
Instruction stream coherence



But… many fragments should be able to share an instruction stream!

```
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, l(0.0), l(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, l(1.0)
```

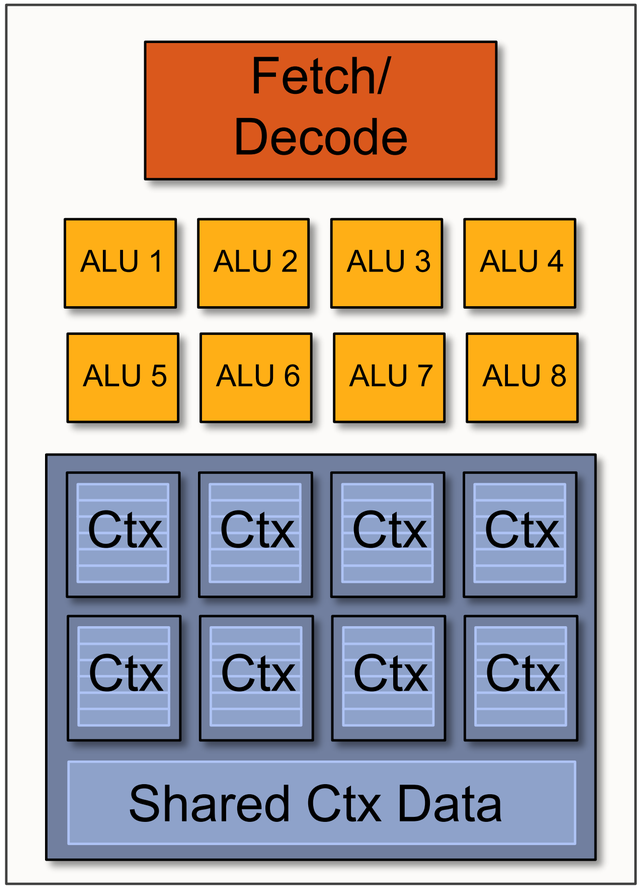


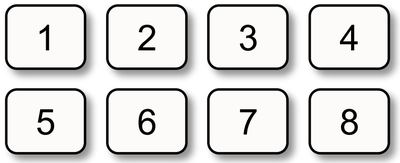


Amortize cost/complexity of managing an instruction stream across many ALUs

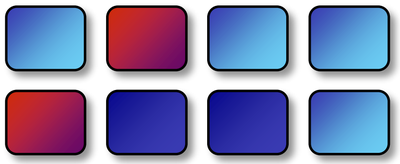
SIMD processing

Single Shader Program Multiple Fragments/Vertices

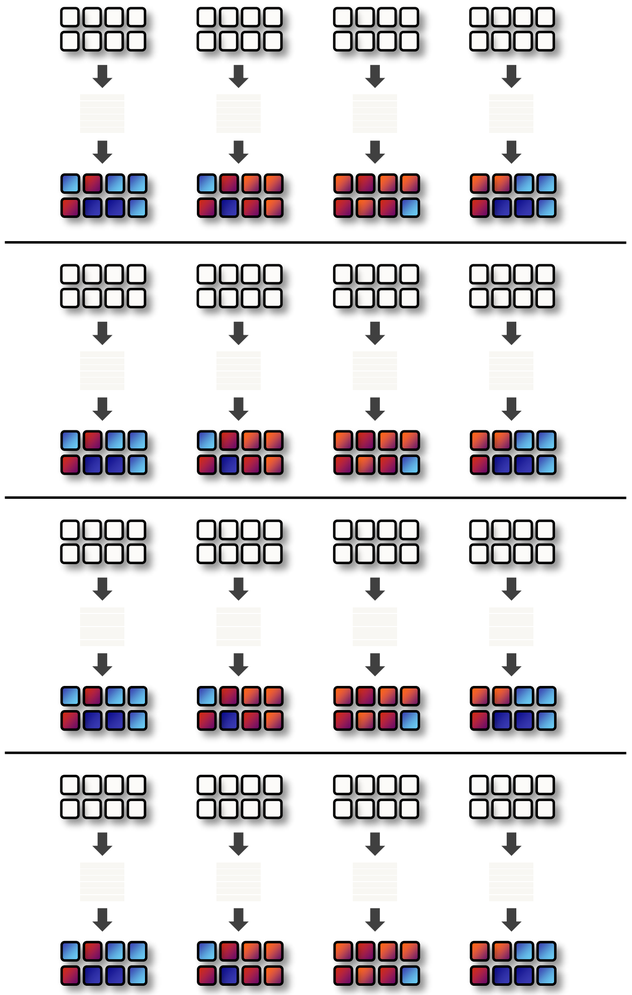


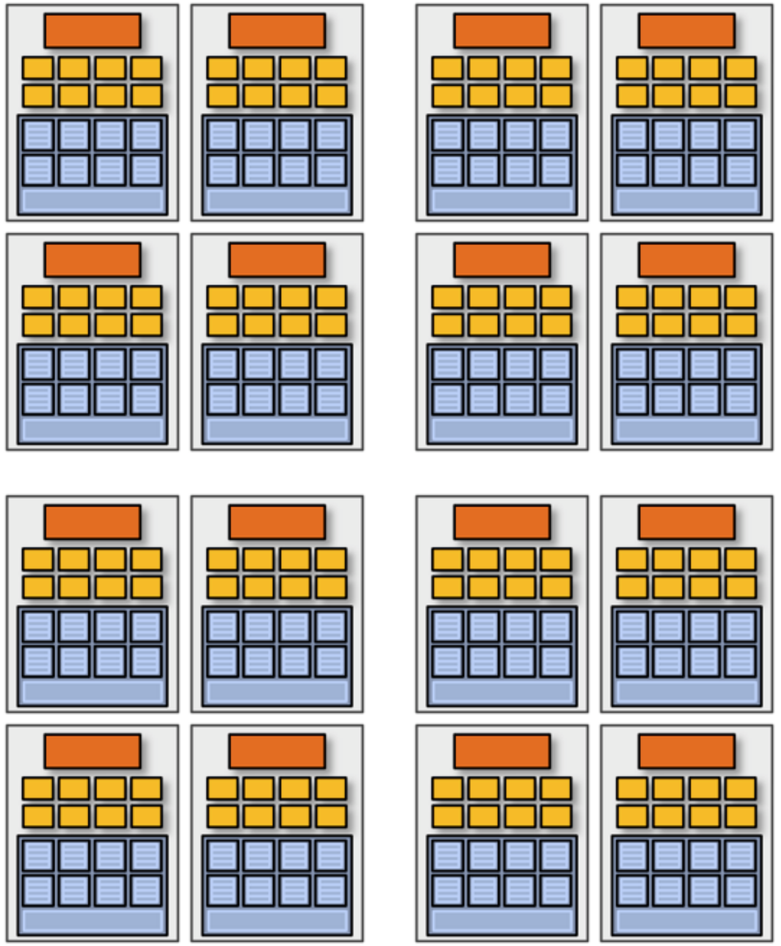



```
<VEC8_diffuseShader>:
VEC8_sample vec_r0, vec_v4, t0, vec_s0
VEC8_mul  vec_r3, vec_v0, cb0[0]
VEC8_madd vec_r3, vec_v1, cb0[1], vec_r3
VEC8_madd vec_r3, vec_v2, cb0[2], vec_r3
VEC8_clmp vec_r3, vec_r3, l(0.0), l(1.0)
VEC8_mul  vec_o0, vec_r0, vec_r3
VEC8_mul  vec_o1, vec_r1, vec_r3
VEC8_mul  vec_o2, vec_r2, vec_r3
VEC8_mov  vec_o3, l(1.0)
```

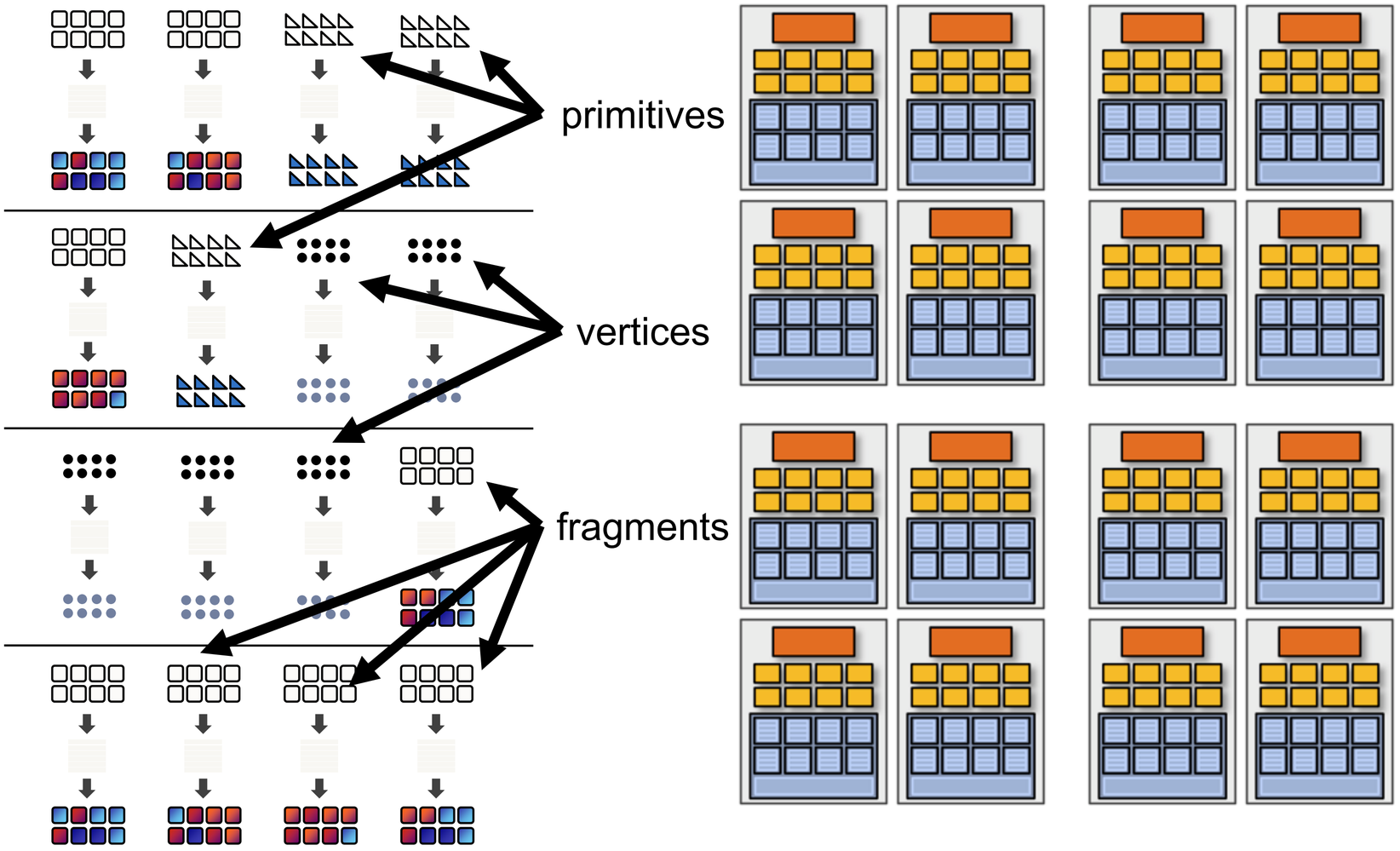


16 SMs, each with 8 ALUs. Each SM runs the same program (fragment shader)



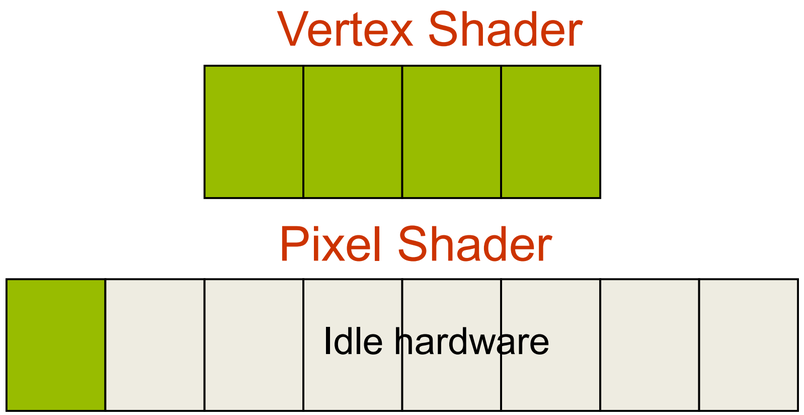


16 SMs, each with 8 ALUs. Each SM runs a different program (shader)



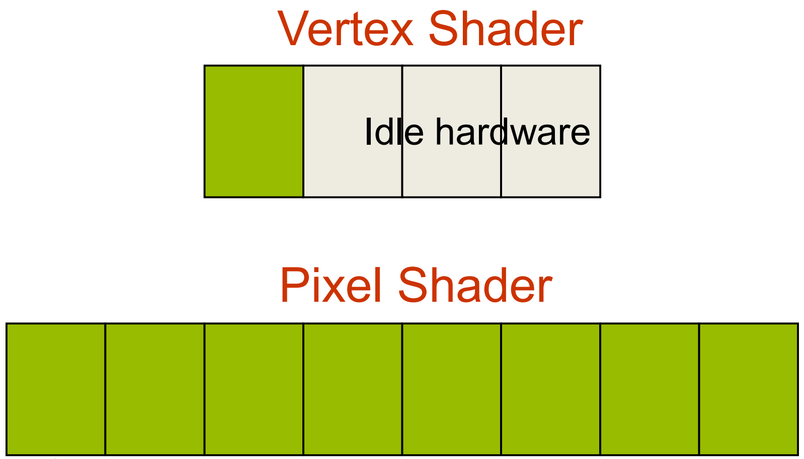
Vertex and Fragment Processing Share Unified Processing Elements

- Load balancing HW is a problem



Heavy Geometry

Workload Perf = 4

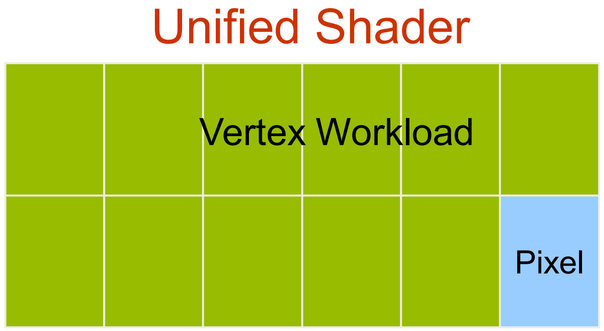


Heavy Pixel

Workload Perf = 8

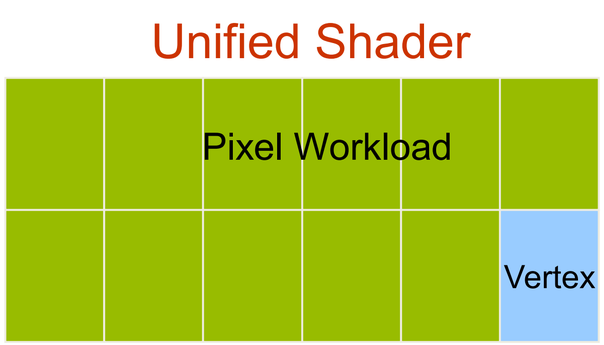
Vertex and Fragment Processing Share Unified Processing Elements

- Load balancing SW is easier



Heavy Geometry

Workload Perf = 11



Heavy Pixel

Workload Perf = 11