

Realistic Image Synthesis

Tutorial Week #1

September 2018

Topics for today

- I. Camera view
- II. Rendering loops
- III. Getting started with the assignments



Teaching assistants



Joey

Keven

Nicolas

{joey.litalien, keven.villeneuve, nicolas.vibert}@mail.mcgill.ca

Tutorials

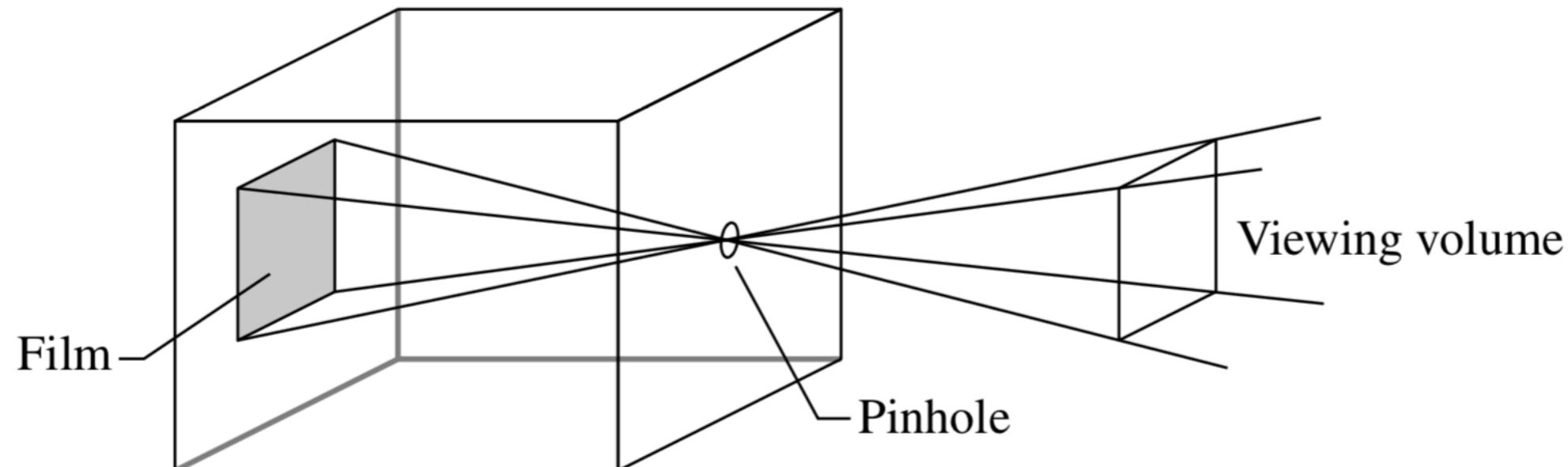
- ▶ Complement the lectures.
- ▶ Answer your questions.
- ▶ Help you with the assignment code.
- ▶ A different TA will be assigned for each assignment, but other TA's may help during debugging sessions.

Camera view

Quick review from lectures

Pinhole camera

- ▶ Infinitely small aperture, hence the name.
- ▶ Defined with position, up vector and look-at vector.



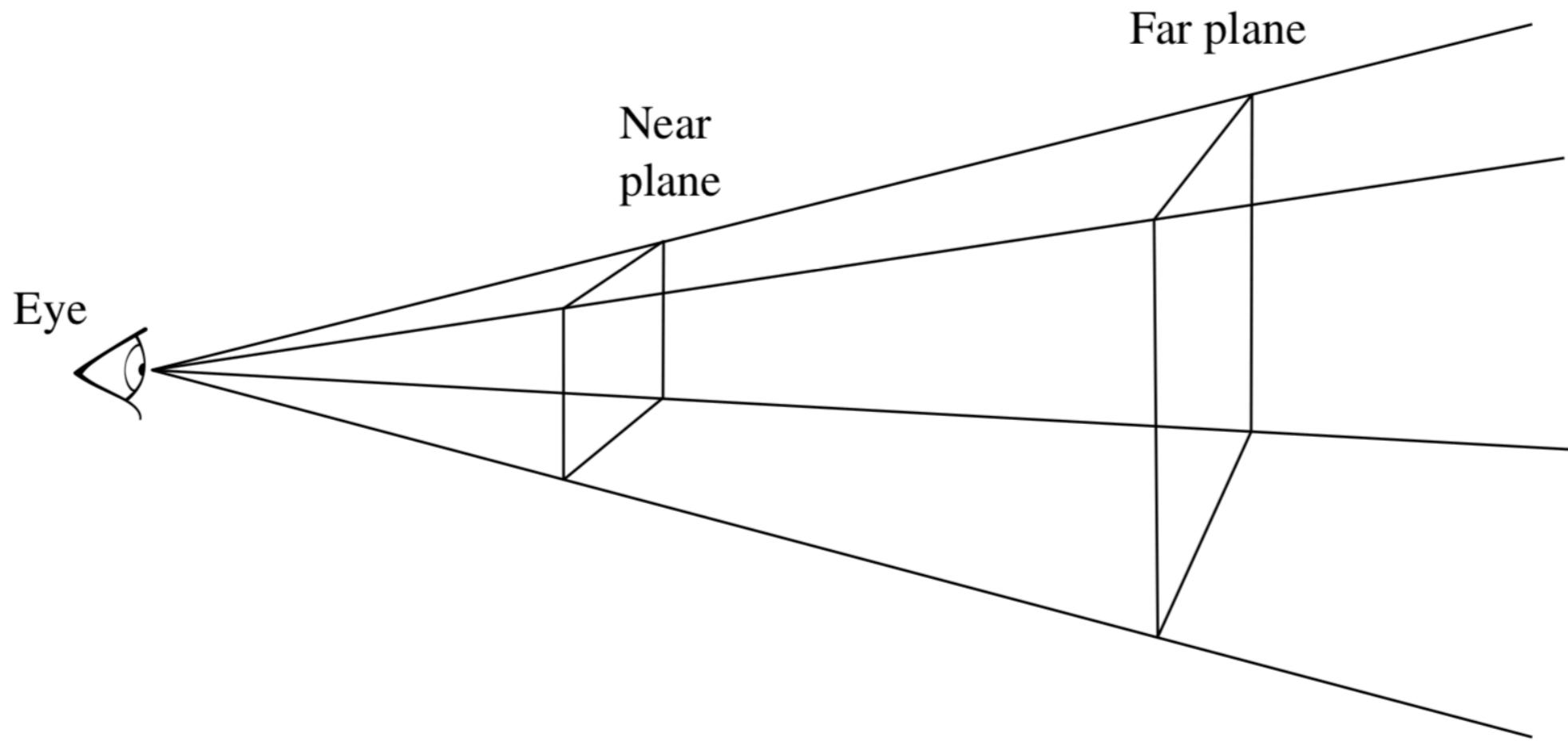
Pinhole camera IRL

- ▶ Hole in a cardboard box to see solar eclipses.



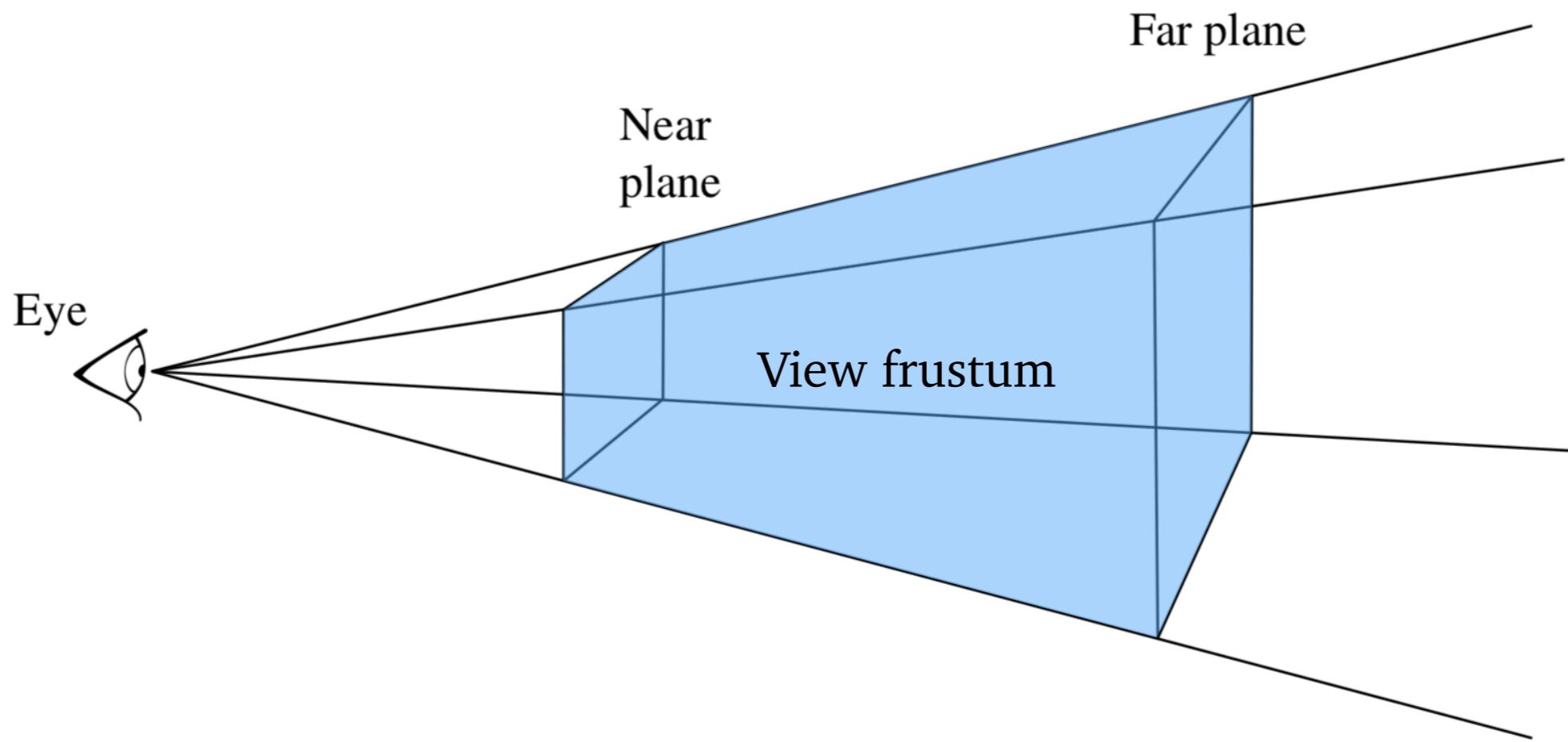
Pinhole camera

- ▶ Simulate pinhole by placing film *in front of* hole.
- ▶ *Hole* is what we call *eye* in rendering.



Pinhole camera

- ▶ Simulate pinhole by placing film *in front of* hole.
- ▶ *Hole* is what we call *eye* in rendering.



Coordinate spaces

- ▶ **Object space**

Coordinate system in which geometric primitives are defined.

- ▶ **World space**

Standard frame that all other spaces are defined in terms of.
Each primitive has an object-to-world transformation.

- ▶ **Camera space**

New coordinate system with origin at camera eye.

Negative z -axis = Viewing direction.

y -axis = Up direction.

Camera matrices

- ▶ **Model matrix**

Acts on the vertices of the object in local coordinates.

Model = Translation · Rotation · Scale (of your actual mesh).

- ▶ **View matrix**

World-to-camera transform that puts the camera at the center of the space.

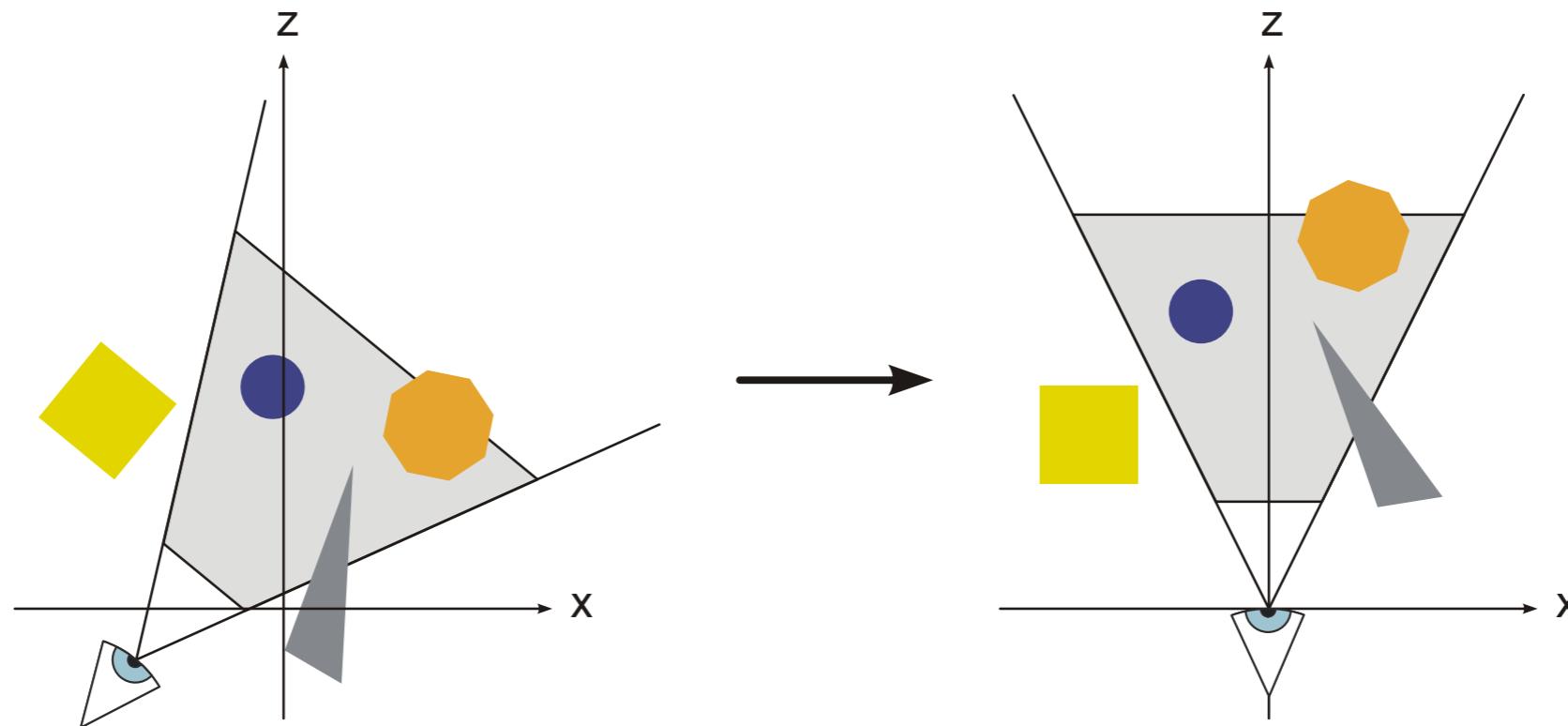
- ▶ **Projection matrix**

Perspective transform from camera space to clip coordinates.

Simulates how we perceive things IRL.

View matrix

- ▶ Moves the world so that camera is at origin.
- ▶ Expressed as a 4×4 transformation matrix.

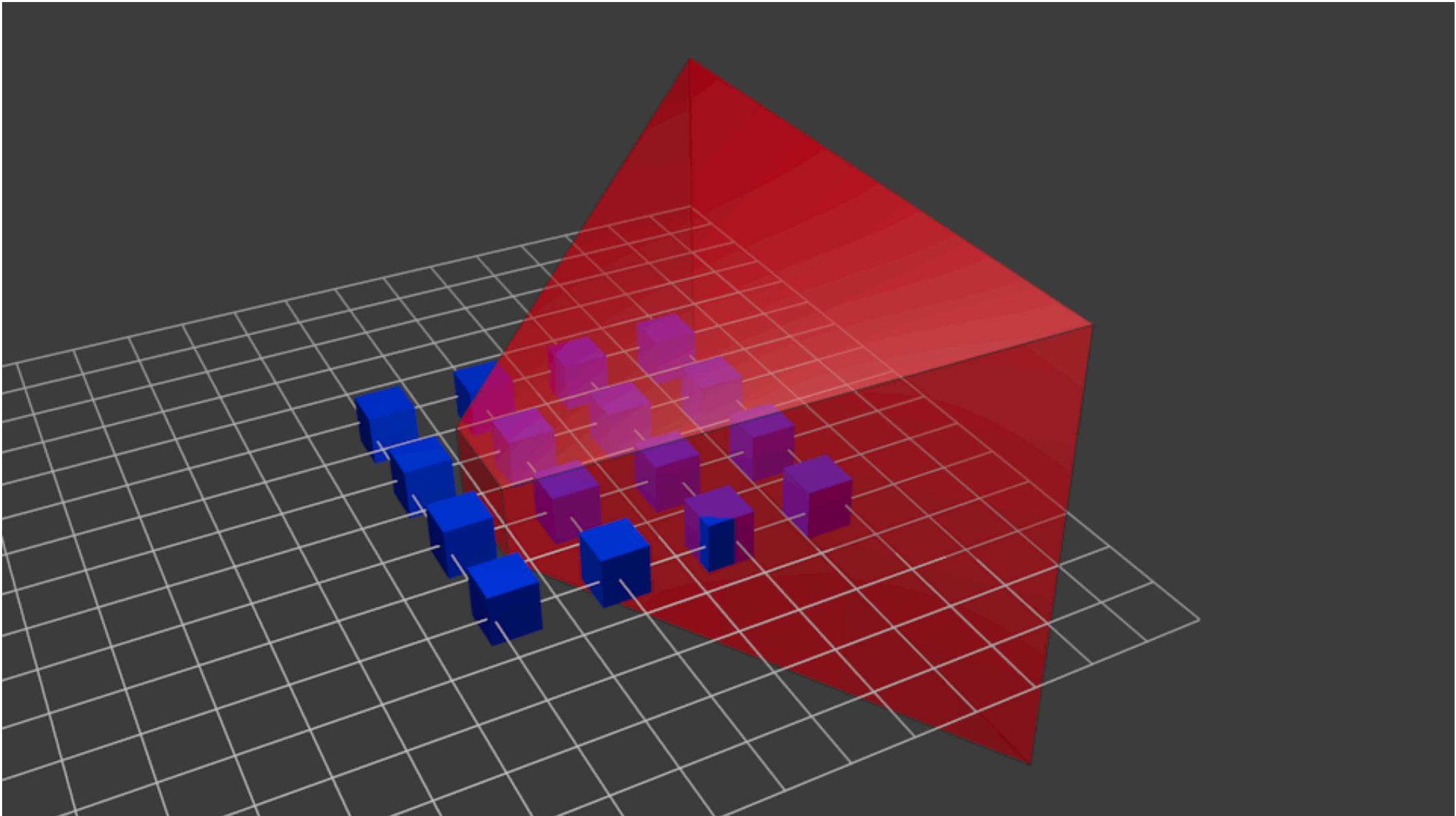


Projection matrix

- ▶ Transforms view frustum (truncated pyramid) into unit cube.
- ▶ Expressed as a 4×4 projection matrix.
- ▶ After transformation, models are said to be in *clip coordinates*.

Camera view

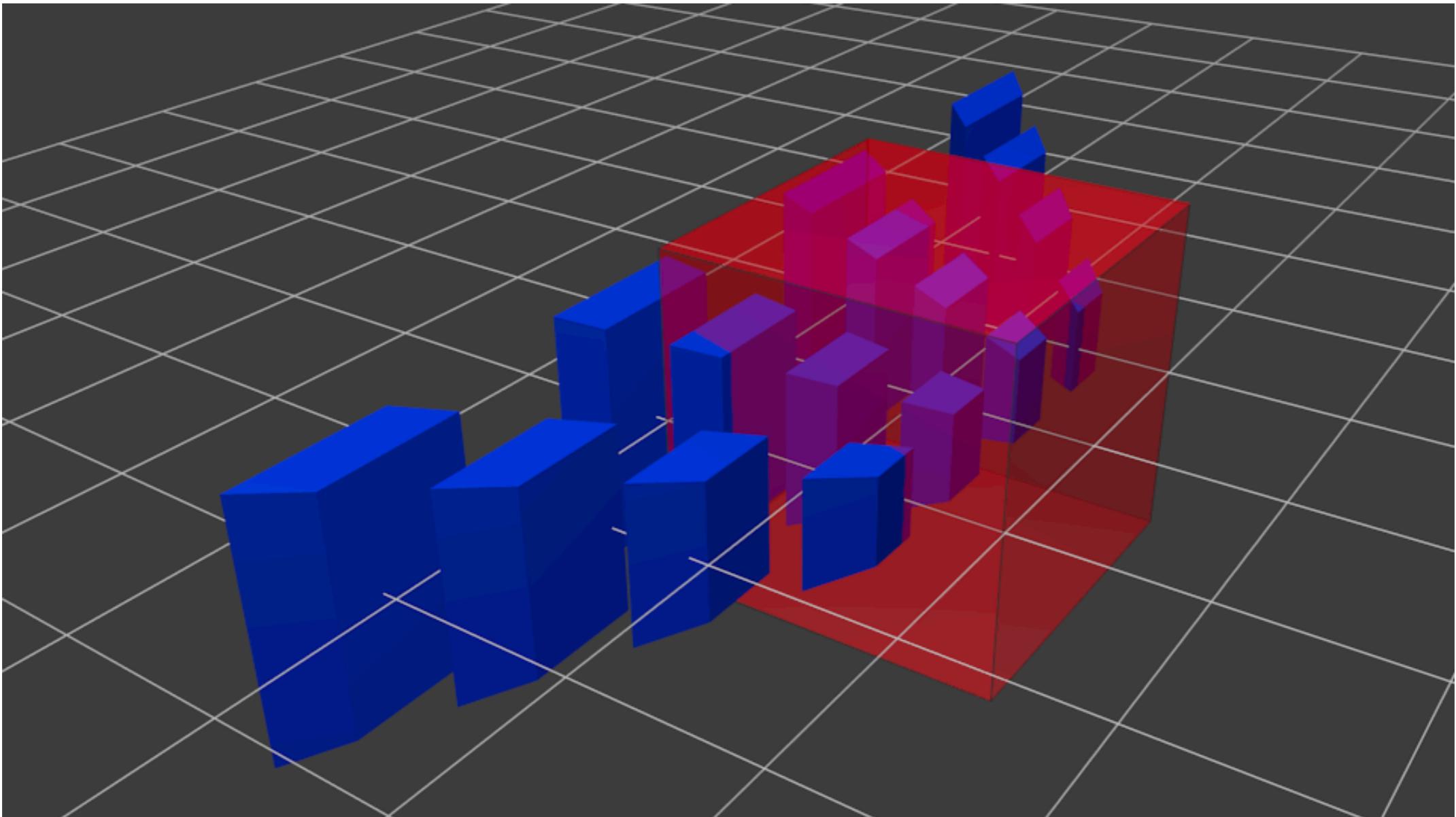
Projection matrix



Before (camera space)

Camera view

Projection matrix



After (clip coordinates)

Further readings

- ▶ **Offline**

PBRT3 Chapter 6.1 *Projective camera models.*

- ▶ **Real-time**

RTR4 Chapter 4.7 *Projections.*

Good tutorial (includes the previous images for the projection matrix):

<http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices>

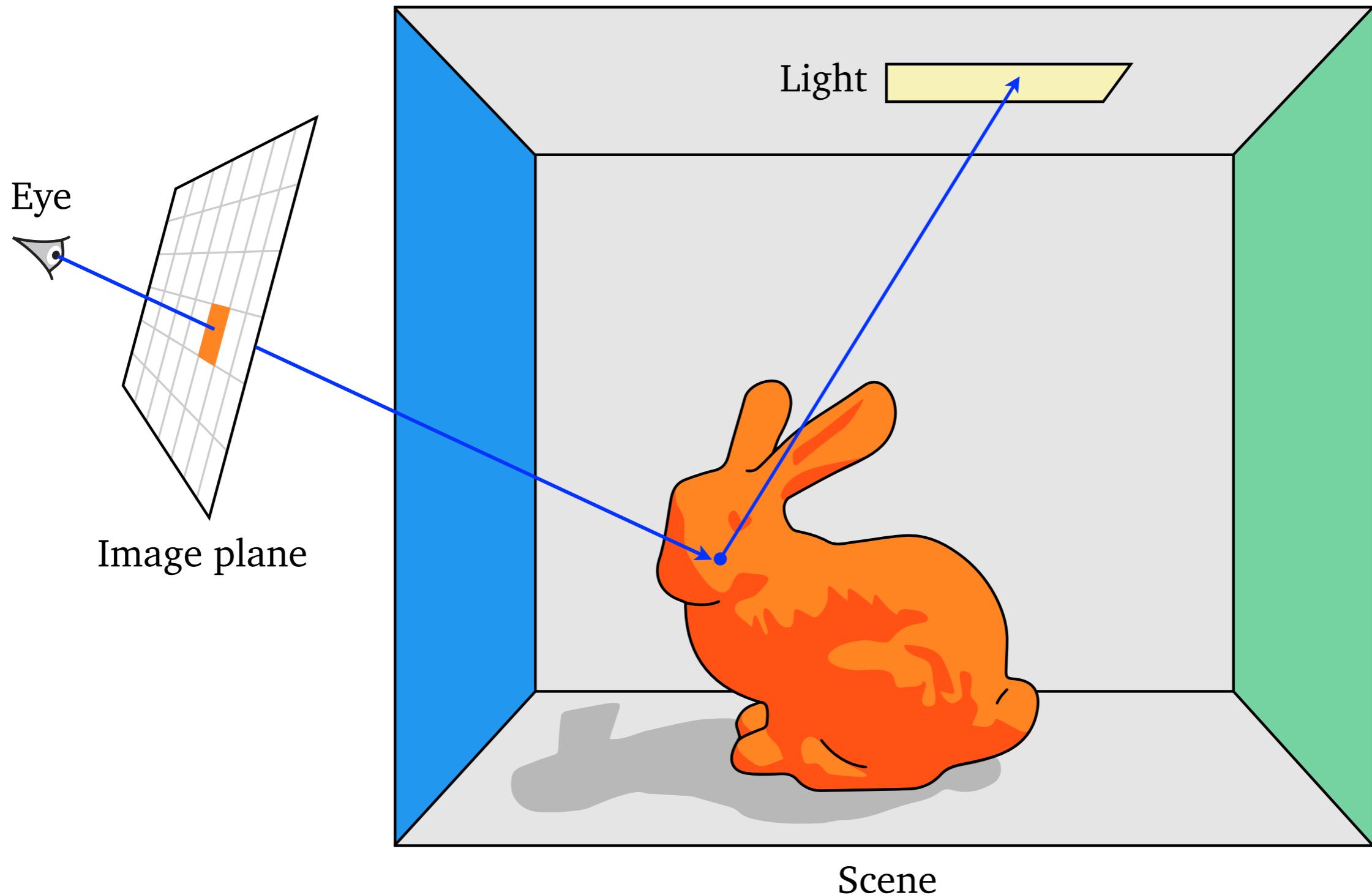
Offline rendering loop

Ray tracing 101

1. Shoot rays from eye through each pixel of the image plane.
2. Hit a surface, retrieve its reflectance property (normal, albedo, etc.).
3. Evaluate light contribution (and possibly recurse).
4. Profit.

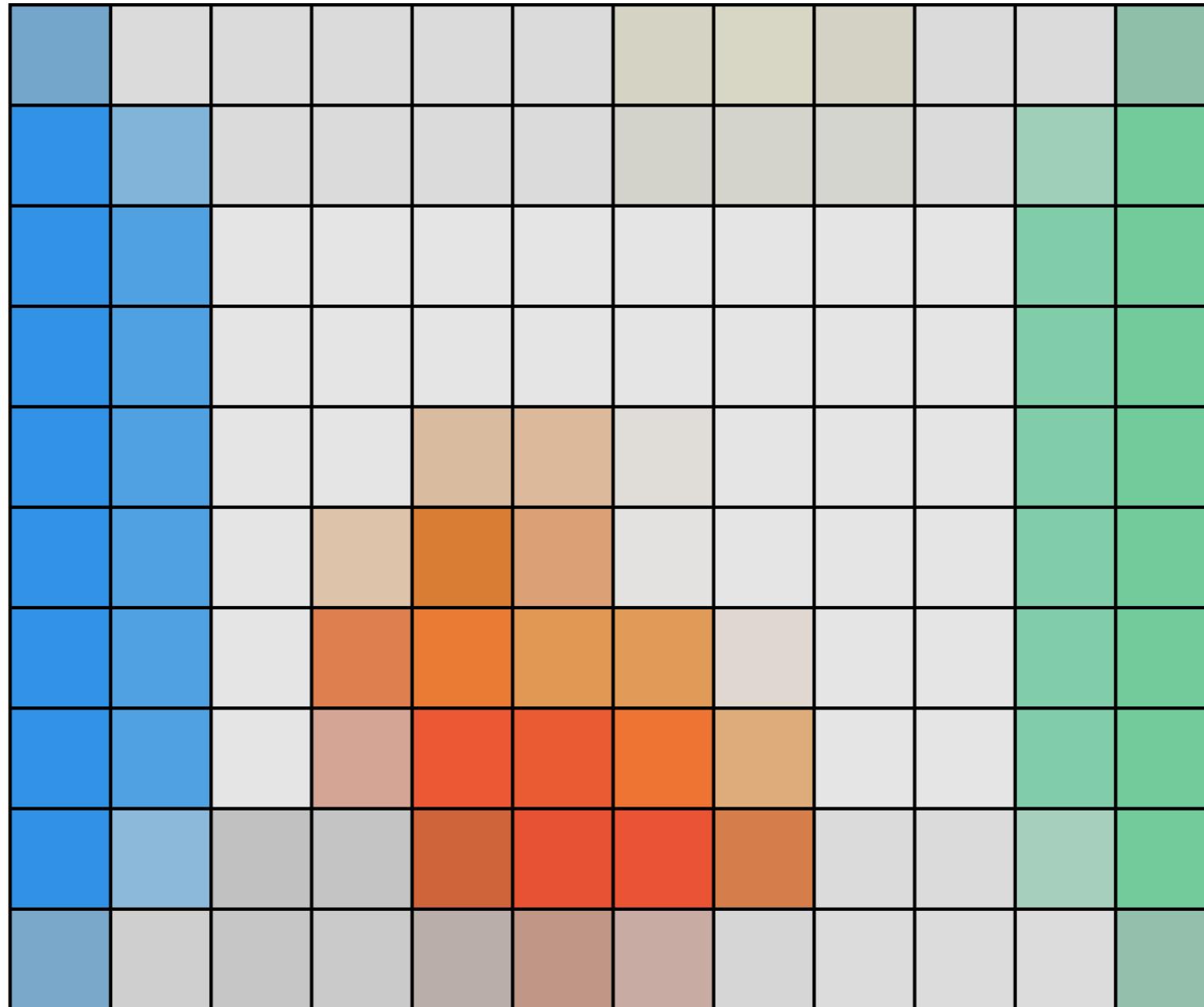
Offline rendering loop

Ray tracing 101



Offline rendering loop

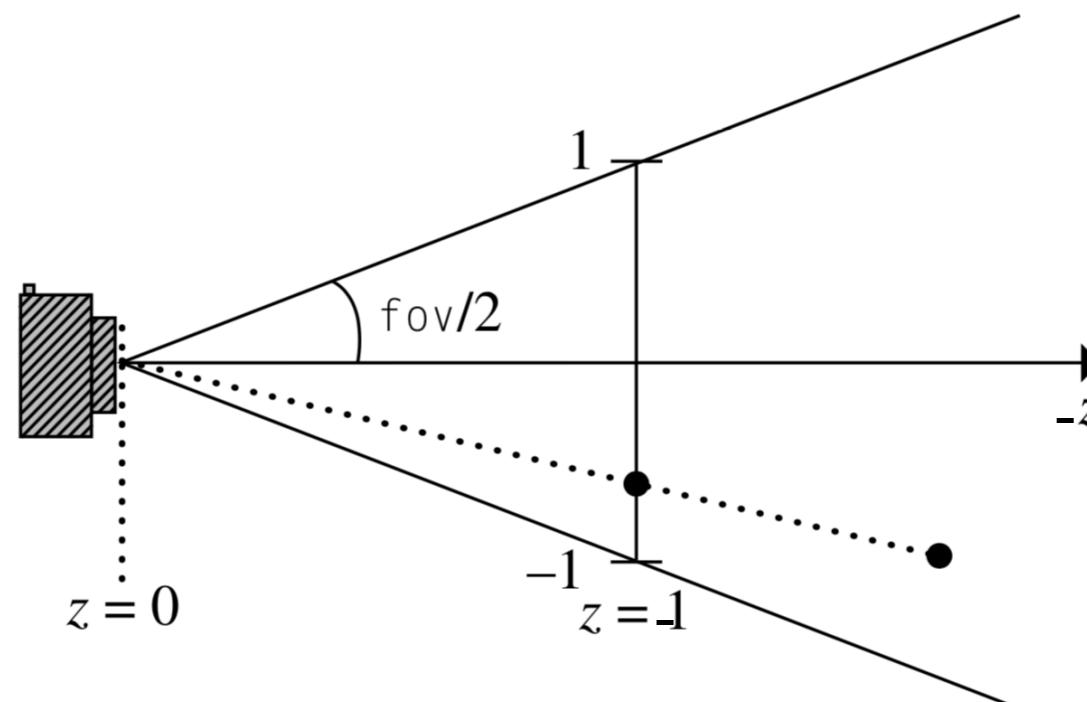
Ray tracing 101



Low-res image (after ray traced)

Field of view

- ▶ Pixel location should be in the interval $[-1, 1]$.
- ▶ Account for FOV by scaling the pixel location:
$$\text{scaling} = \tan(\text{deg2rad}(\text{fov}) / 2 \cdot f);$$
- ▶ Adjust film dimension by scaling x -axis by aspect ratio (w/h).



Field of view

- ▶ TinyRender uses *vertical* FOV.
- ▶ Seems counterintuitive as we usually prefer landscape mode in photography.
- ▶ Arbitrary choice to match camera settings of reference scenes, so just roll with it.

View matrix

- ▶ Use OpenGL Mathematics:

```
glm::mat4 inverseView = glm::lookAt(eye, at, up);
```

- ▶ Gives you the matrix in *row major order*.
- ▶ To map a ray from camera to world spaces:
 1. Augment to homogeneous coordinates (4D).
 2. Multiply by inverse view matrix *on the right*.
- ▶ Row major form is more efficient computationally;
Alternative: use `glm::transpose()` and multiply *on the left*.

Offline rendering loop

Ray tracing pseudocode

```
// Compute camera settings
// Clear image RGB buffer & instantiate sampler

for (x = 0; x < width; ++x) {
    for (y = 0; y < height; ++y) {
        (px, py) = getPixelLoc(x, y, width, height);
        dir = toWorld(px, py);
        ray = Ray(camera.o, dir);
        pixelColor = render(ray, sampler);
        splatOnScreen(x, y, pixelColor);
    }
}
```

Offline rendering loop

Ray tracing pseudocode

```
// Compute camera settings  
// Clear image RGB buffer & instantiate sampler  
  
for (x = 0; x < width; ++x) {  
    for (y = 0; y < height; ++y) {  
        (px, py) = getPixelLoc(x, y, width, height);  
        dir = toWorld(px, py);  
        ray = Ray(camera.o, dir);  
        pixelColor = render(ray, sampler);  
        splatOnScreen(x, y, pixelColor);  
    }  
}
```

Generating rays

- ▶ Different ways to sample ray through pixel:
 - Single ray through pixel center. (A1)
 - Single ray through random position on pixel.
 - Multiple rays through pixel center.
 - Multiple rays through random position on pixel. (A1 bonus)



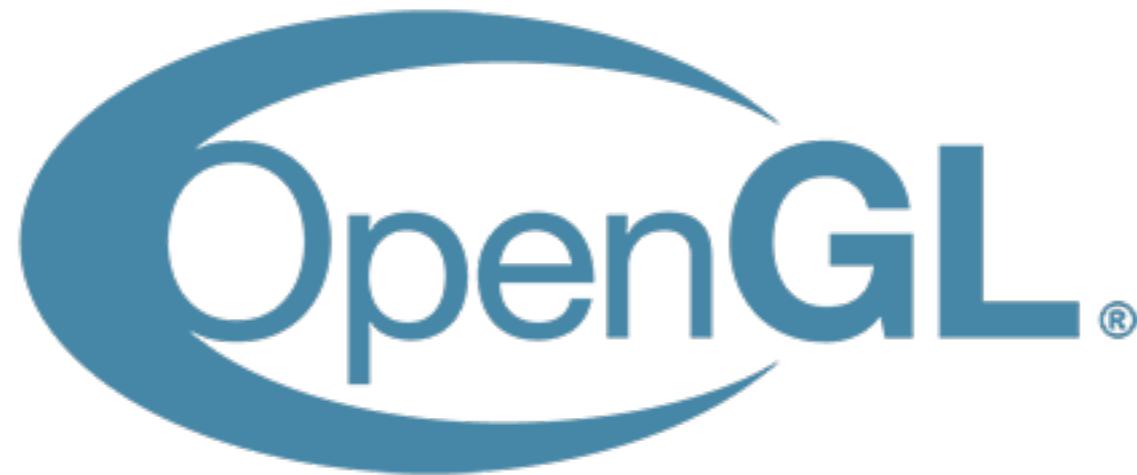
Real-time rendering loop

OpenGL 101

- ▶ OpenGL is an API to access the GPU.
- ▶ This course uses a relatively modern version (3.3).
- ▶ Good references:
 - Official docs: <http://docs.gl>
 - Shaderific: <http://www.shaderific.com/glsl>

OpenGL 101

- ▶ GL = State Machine + Buffers.
- ▶ Data is passed through *buffers*.
- ▶ Commands are then used to change the *state* of the machine.



Modern GL pipeline

1. Initialize buffers.
2. For each frame:
 - a) Bind buffers (tell GPU which VAO/VBO to use).
 - b) Update uniforms.
 - c) Draw call.
3. Clean up.

GL Buffers

- ▶ **Screen buffer**

Where all the pixels are written.

- ▶ **Vertex Buffer Object (VBO)**

Where vertex attributes (position, normal, color, etc.) are stored.

- ▶ **Vertex Array Object (VAO)**

- ▶ **Other buffers**

Framebuffer Object (FBO)

Buffer Texture

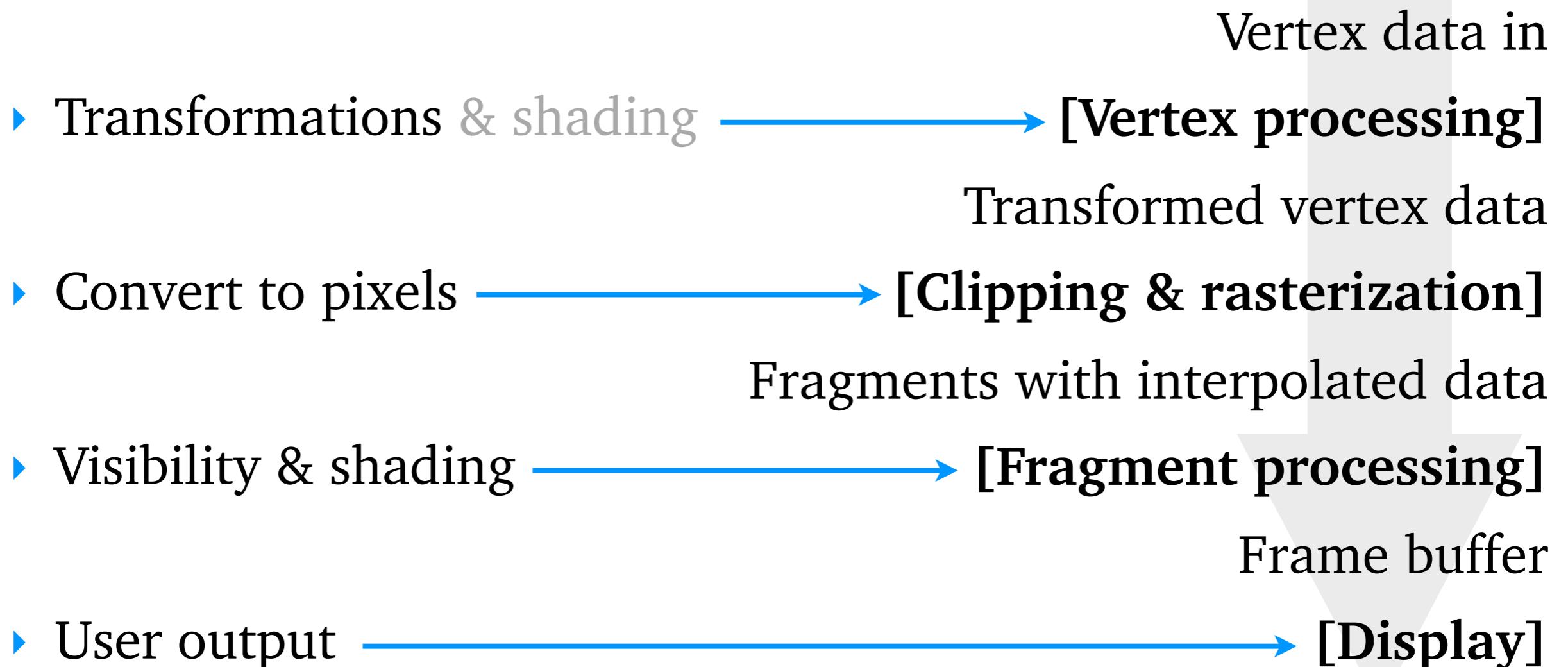
GL Buffers

- ▶ They are used the same way:
 1. Create buffer.
 2. Bind buffer (to use it).
 3. Fill buffer with data.

Rasterization

- ▶ Process of converting a continuous object to a discrete representation on a raster (pixel) grid.
- ▶ At a high-level:
 1. Take triangles.
 2. Figure out which pixels they cover.
 3. Perform some computation on these pixels.

Programmable rasterization pipeline



So, what do I code?

- ▶ Render pass that actually calls the shaders.
- ▶ GLSL shaders:
 - Vertex shader (.vs file).
 - Fragment shader (.fs file).

Vertex shader

- ▶ Done at every vertex (hence the name).
- ▶ Means that it can manipulate attributes of vertices (position, normal, etc.).
- ▶ Comes earlier in the graphics pipeline.

Fragment shader

- ▶ Comes after the vertex shader.
- ▶ And *after* the rasterization step.
- ▶ Takes care of how the pixels between the vertices look.
- ▶ Interpolated between the defined vertices following specific rules to *fill in* the fragments.

Shader logic

- ▶ Vertex shader will relay some information to the fragment shader (out).
- ▶ **Vertex shader**
Transform triangles to match camera view by setting `gl_Position`, and retrieve necessary information for shading.
- ▶ **Fragment shader**
Compute the actual color of the pixel using whatever info passed from the vertex shader.

MVP Matrix

- ▶ Use GLSL uniforms:

```
gl_Position = projection * view * model  
* vec4(position, 1.0);
```

- ▶ That's it. Field of view is handled by the code for the real-time part.

TinyRender

General information

- ▶ Minimalistic rendering engine written in C++11.
- ▶ Cross-platform (Windows, Linux and macOS).
- ▶ Both offline and real-time rendering frameworks.
- ▶ Written by us (the TA's), so hopefully it works!
- ▶ First time using it, so signal any important bug by email if you find one.

What you'll be able to render



Rendered with path tracer from A5

And with a few tweaks...

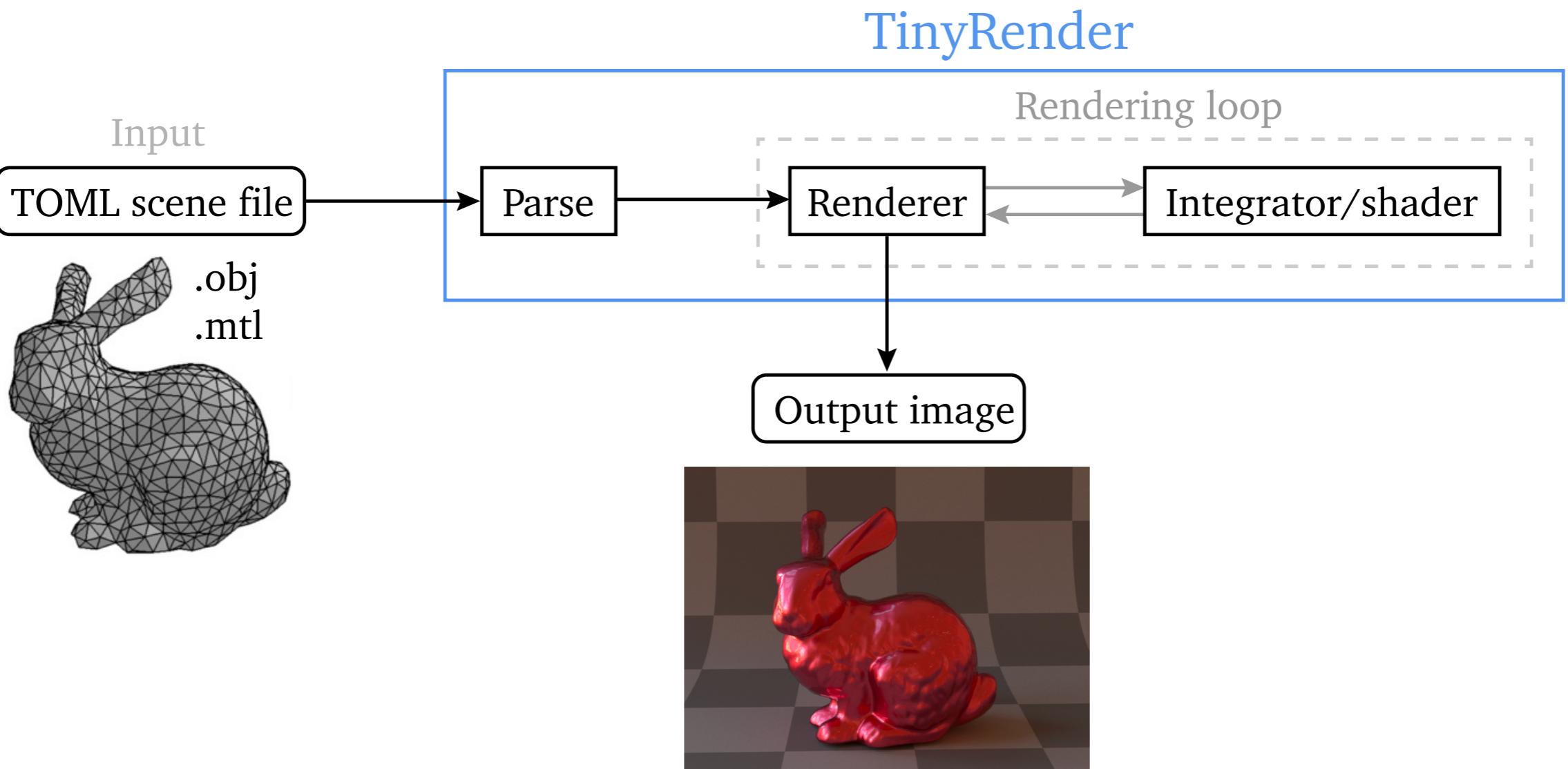


Gold



Brushed glass

Overview of the codebase

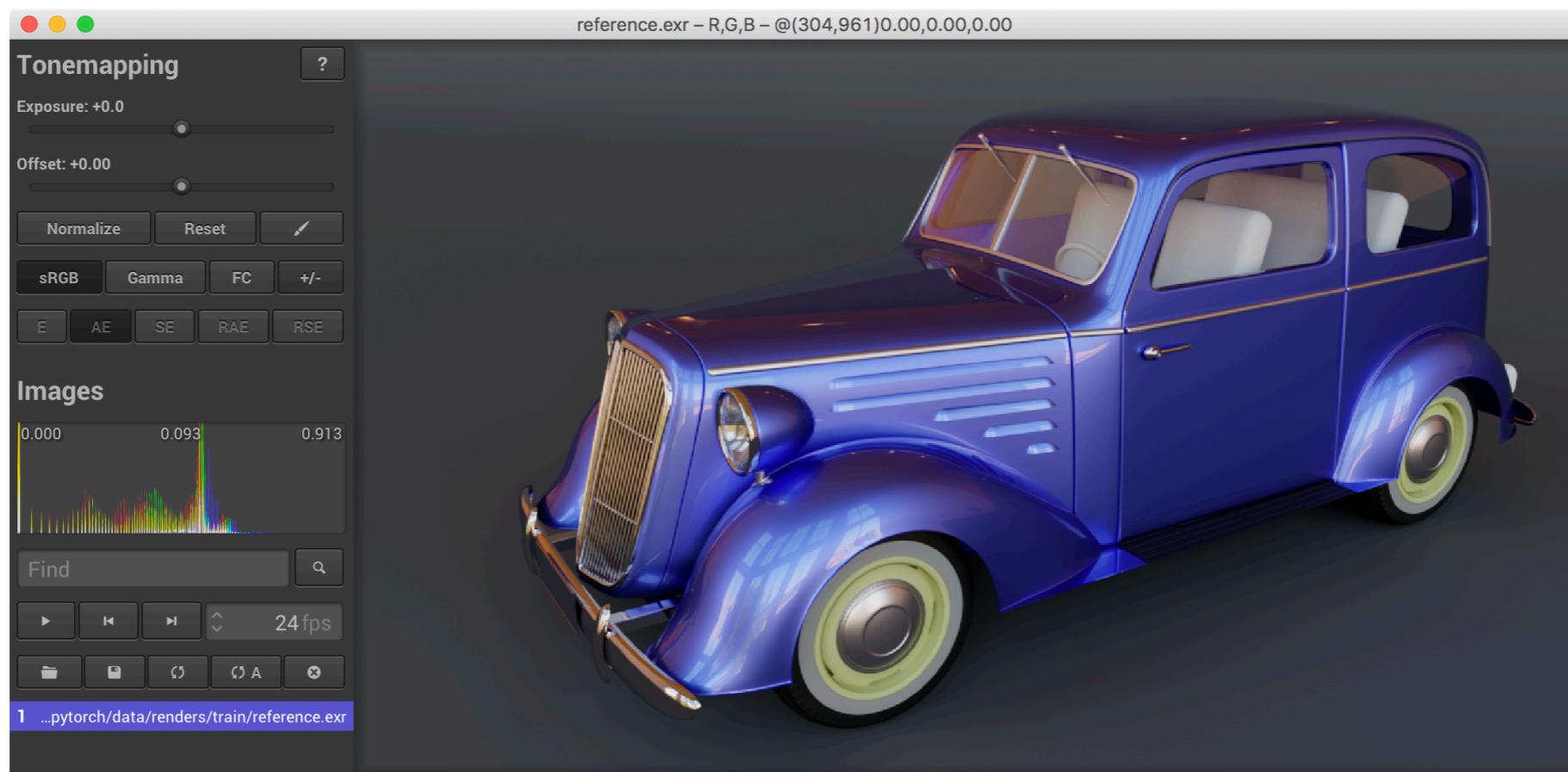


Building instructions

- ▶ All details can be found in the assignment handout.
- ▶ Requires a few things to be installed beforehand.
- ▶ If you want to compile TinyRender in the command-line under Ubuntu 14, you have every right to do so.
- ▶ **But**, we are *not* going to help you.

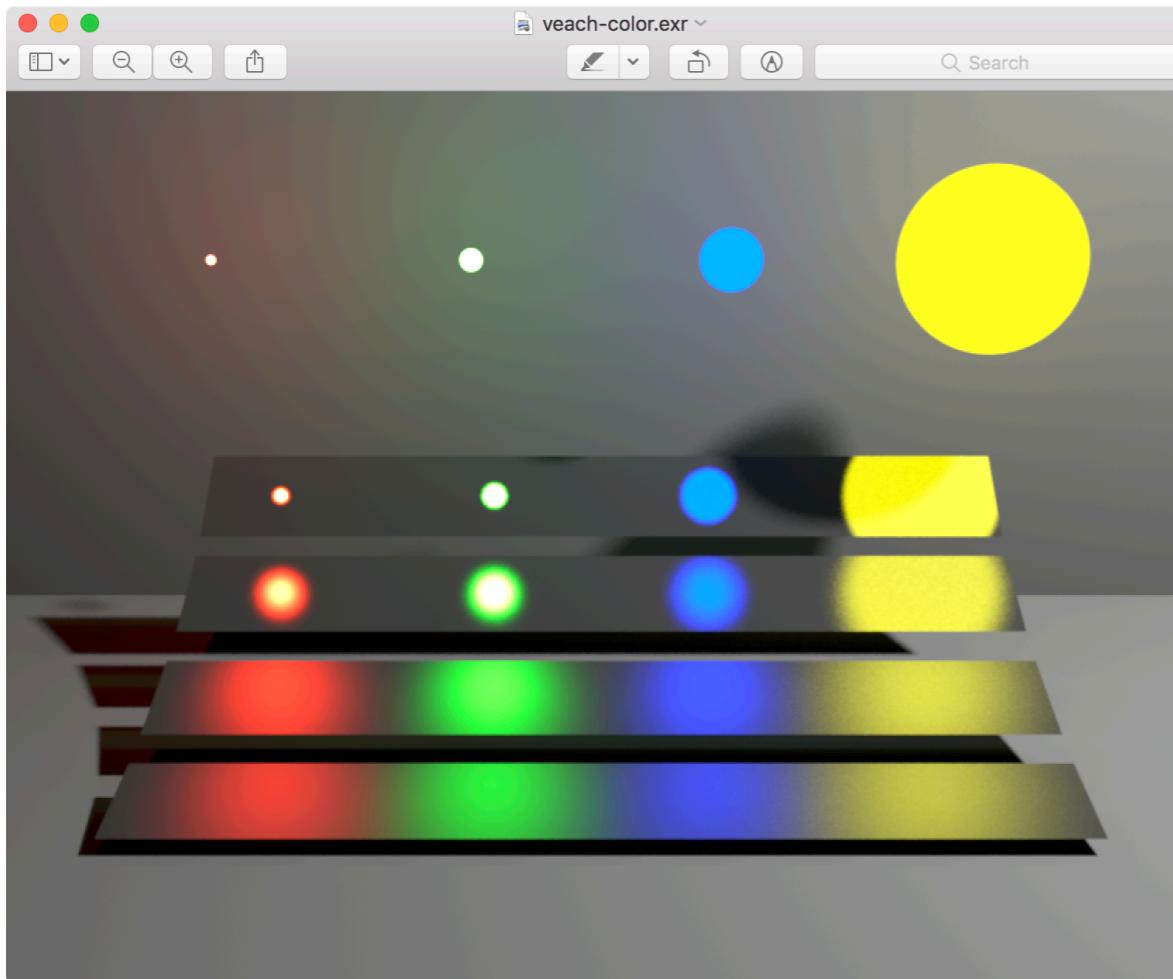
High dynamic range viewer

- ▶ Offline renders are saved in HDR formats (.exr, .ppm).
- ▶ Requires HDR viewer software (e.g. tev, HDRView).

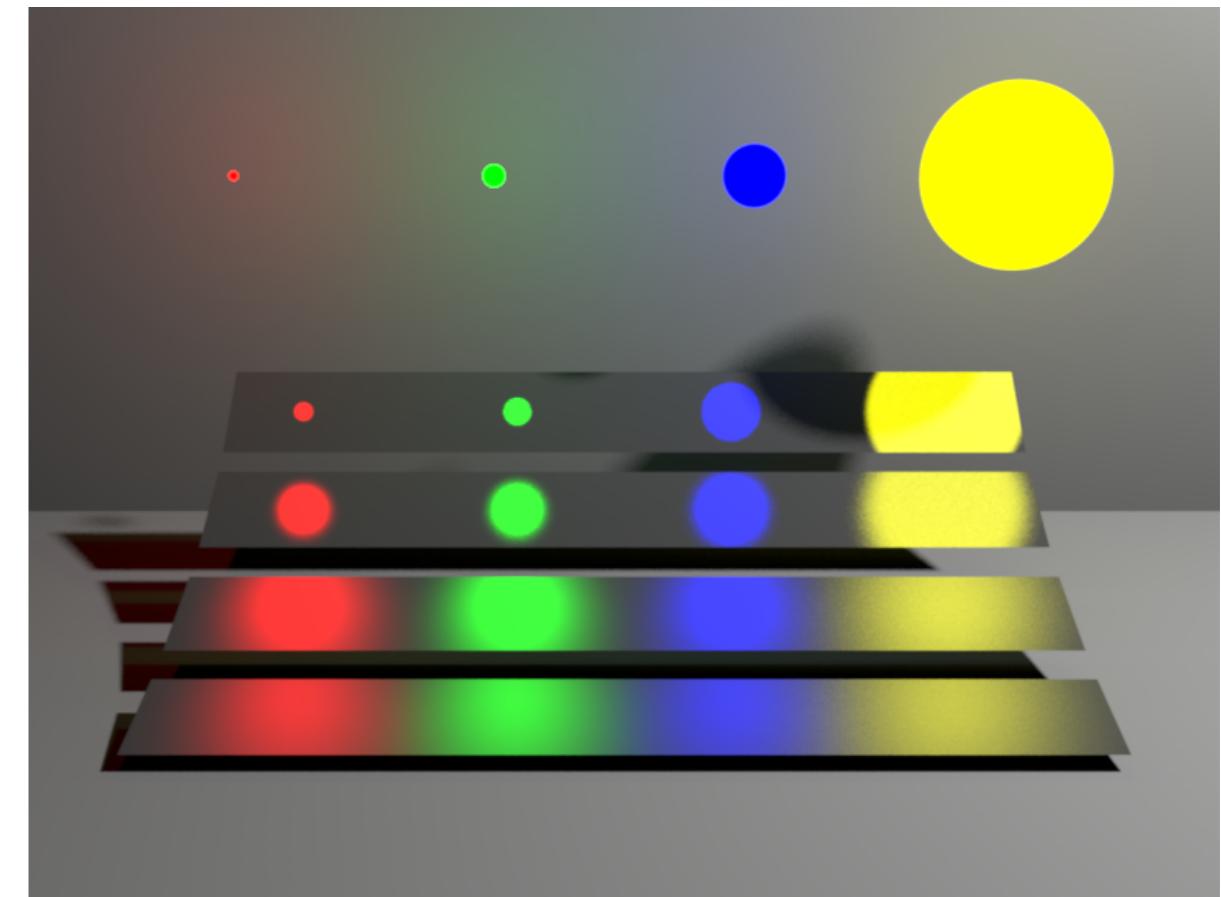


tev

Tonemapping and why it matters



Wrong tonemap (Preview on macOS)



Correct tonemap

Submission script

- ▶ Every assignment requires you to render a few images (both offline and real-time).
- ▶ Python script will properly tonemap your images and bundle them into a self-contained HTML file.
- ▶ This is the file you submit, along with your HDR files and your code.
- ▶ Test submission script beforehand to avoid last minute problems!

Submission script demo

Important notes

- ▶ 3000+ lines of code: you will have to learn how to navigate the code base, *don't be scared.*
- ▶ All assignments use TinyRender, and there is very few overlaps between them.
- ▶ Incremental level of difficulty, but some assignments are more important than others. *A1 is one of them.*
- ▶ You are encouraged to discuss the assignments with your peers, but sharing code or images is *strictly forbidden.*

Questions?