

# Lecture 13: Post-processing and Imaging pipeline

DHBW, Computer Graphics

Lovro Bosnar

29.3.2023.

# Syllabus

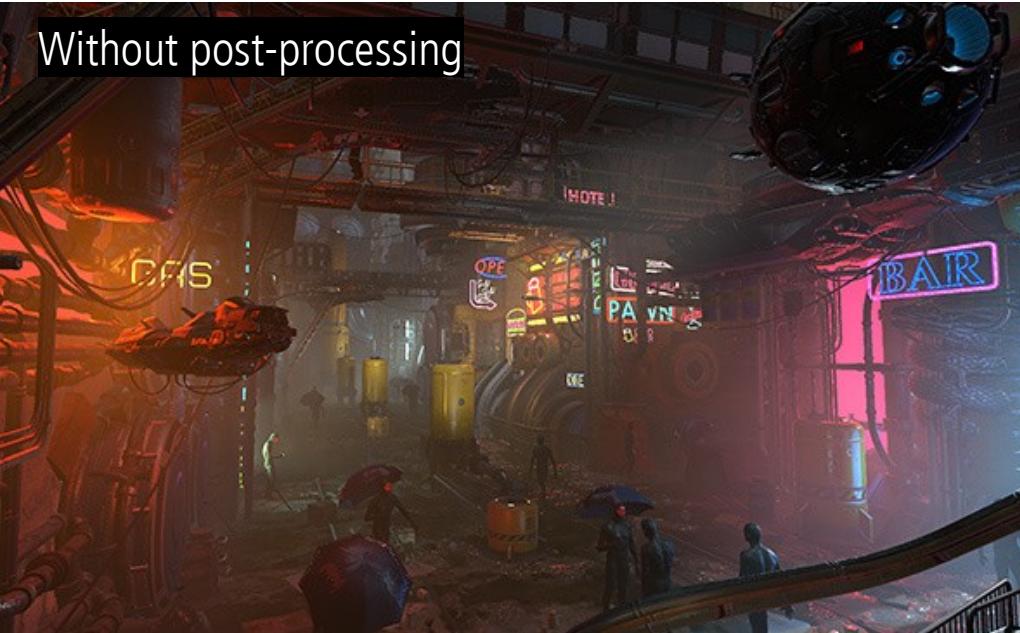
- 3D scene
    - Object
    - Light
    - Camera
  - Rendering
  - Image and display
- Image and display
    - Image-space effects
    - Displaying and storing images
    - Aliasing and anti-aliasing
    - Limitations of display device
- 

# Image-space effects

# Image-space effects

- Rendered image can be directly sent to display or additionally modified before display → **post-processing**
- Post-processing is set of operations performed on a rendered image → **image-space effects**
  - This way, additional effects for achieving realism or artistic styles is possible

Without post-processing

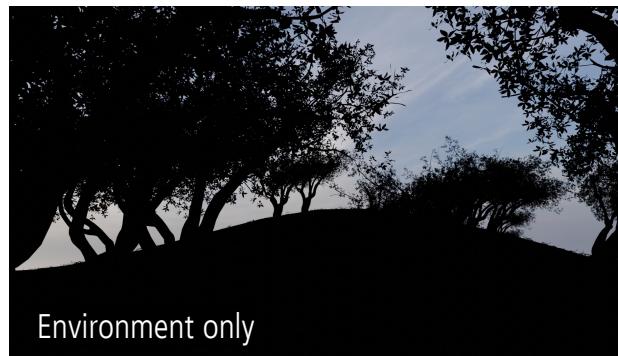
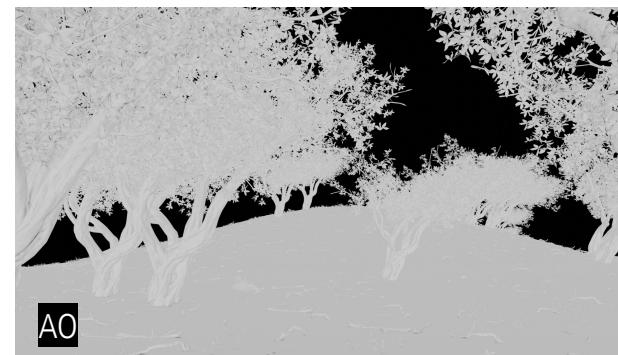
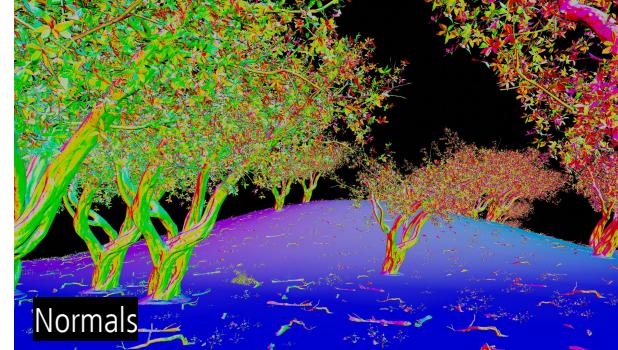


With post-processing



# Post-processing data

- Post-processing of **one frame** can use **multiple buffers (images)** obtained during rendering:
  - Color, diffuse, normal, AO, depth, environment, etc.



# Performing post-processing

- Depending on rendering environment, post-processing of frame can be performed:
  - **Real-time, multi-pass, render-time**
    - 3D scene is rendered to an **offscreen buffer**, such as color (image) buffer
    - Resulting image is treated as image texture which is **applied on screen filling quad**
    - Post-processing is done on this texture **using programmable GPU shaders** (e.g., fragment or compute shaders)
  - **Offline, compositing software**
    - 3D scene is rendered to image file
    - Post-processing is done on images which are stored as separate frames



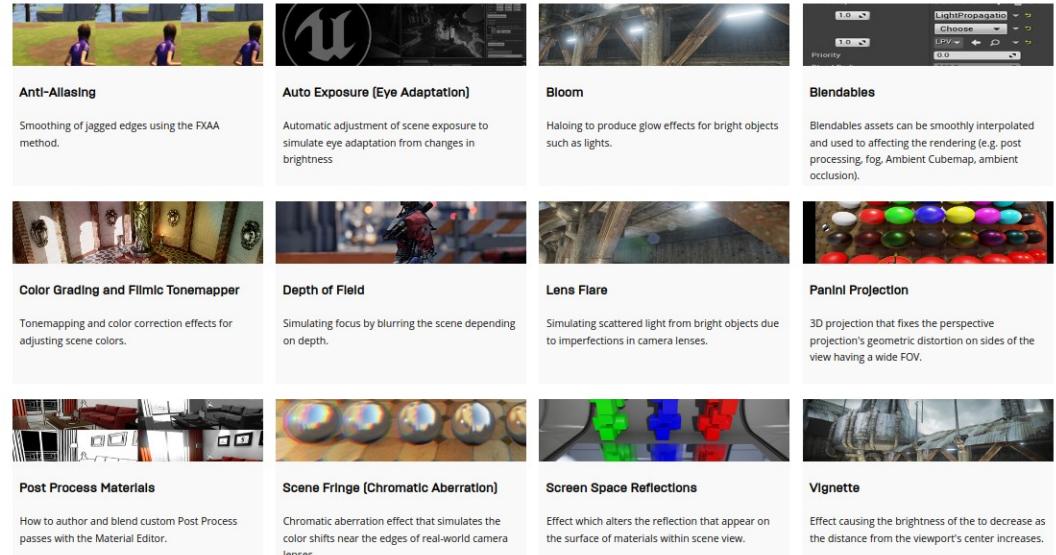
# Post-processing in practice

- Real-time, multi-pass:

- Unity: <https://docs.unity3d.com/Manual/PostProcessingOverview.html>
- Godot: [https://docs.godotengine.org/en/stable/tutorials/shaders/custom\\_postprocessing.html](https://docs.godotengine.org/en/stable/tutorials/shaders/custom_postprocessing.html)
- Unreal: <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/>
- OpenGL: <https://learnopengl.com/In-Practice/2D-Game/Postprocessing>

- Offline:

- Blender: <https://docs.blender.org/manual/en/latest/compositing/introduction.html>
- Nuke: <https://www.foundry.com/products/nuke-family/nuke>
- After Effects: <https://www.adobe.com/products/aftereffects.html>
- Discussion: [https://www.youtube.com/watch?v=7g4xCV0iv4w&ab\\_channel=InspirationTuts](https://www.youtube.com/watch?v=7g4xCV0iv4w&ab_channel=InspirationTuts)



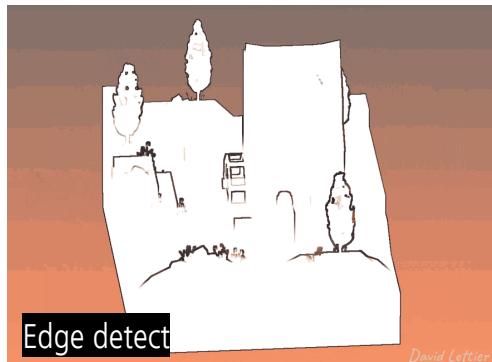
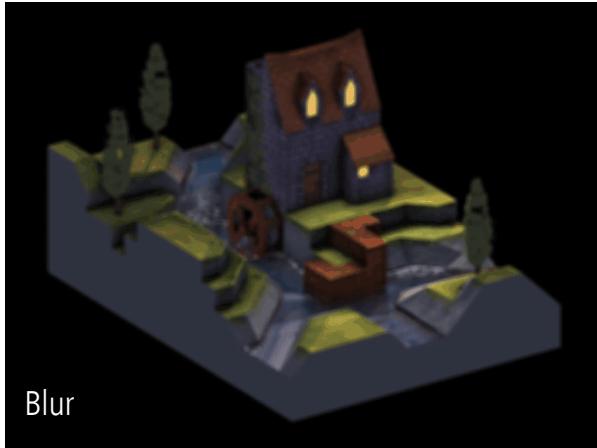
Unreal post-process



Nuke post-process

# Post-processing using image processing

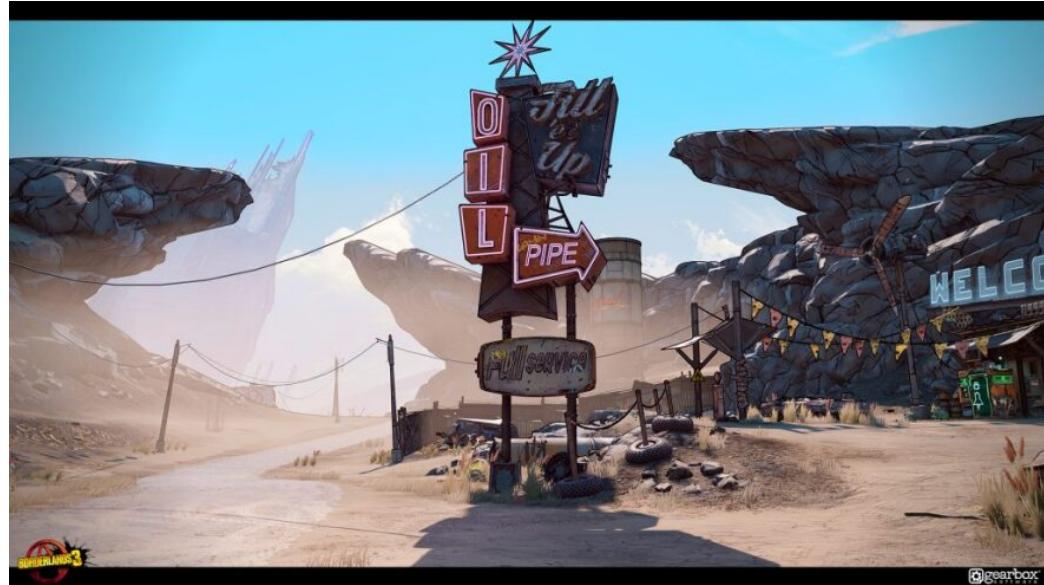
- Rendered image can be post-processed as any other image using image processing techniques (filtering kernels) in shader or CPU.



<https://lettier.github.io/3d-game-shaders-for-beginners/posterization.html>

# Post-processing and NPR

- Often image processing techniques are used to achieve **non-photo realistic rendering effects**



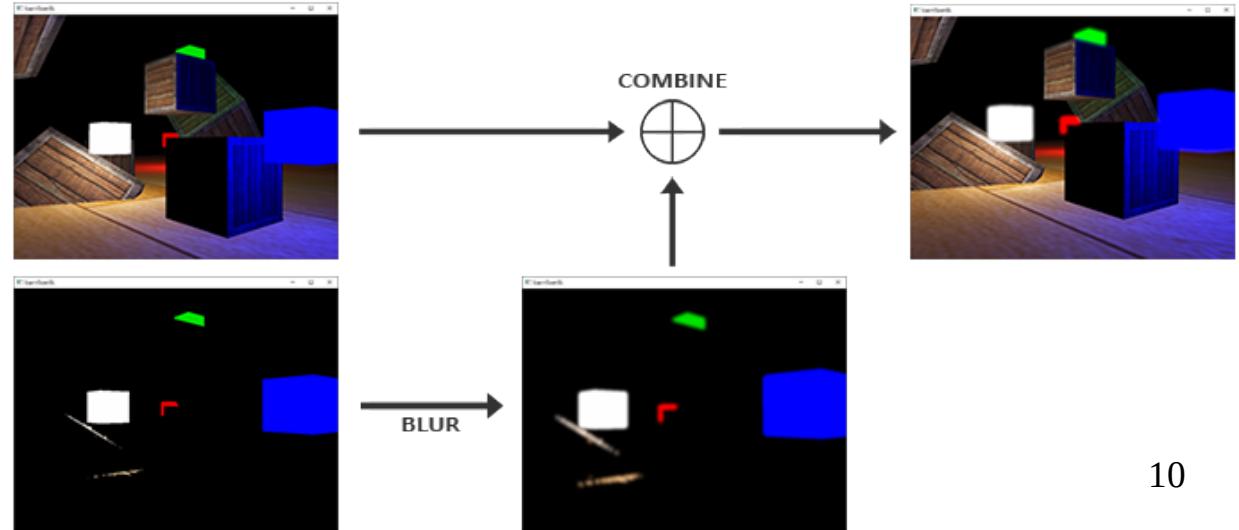
Edge detection (Borderlands)



Edge detection and quantization (Okami)

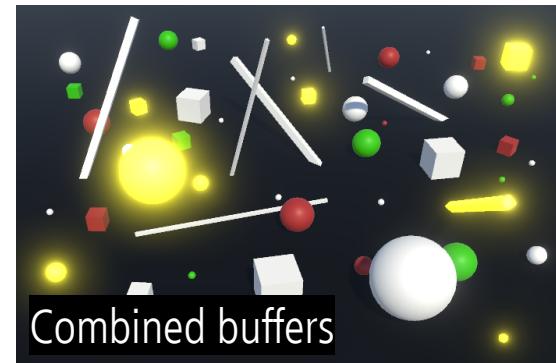
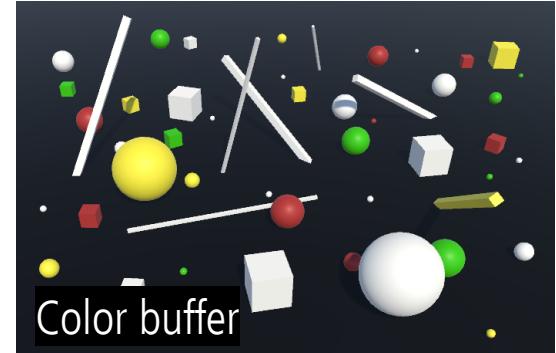
# Post-processing: bloom

- Bloom occurs in camera when charged site in CCD gets saturated and overflows into neighboring sites
- Effect where **extremely bright area spills over adjoining pixels** → making image look overexposed where it is bright
- Concept:
  - Render whole scene in color buffer
  - Render only bright objects into separate buffer
  - Blur buffer with bright objects only (e.g., Gaussian Blur)
  - Combine buffers



# Bloom: examples

- Bloom is one type of **glare effects**. Other glare effects are lens flare: halo and ciliary corona



# Post-processing: lens flare

- **Flare phenomena:** caused by light traveling through a lens system
- Main types: halo (ring around light) and ciliary corona (rays radiating from a point)
  - Dependent on light source position
- Various approaches exist. Examples:
  - Set of squares with different textures, oriented on a line going from the light source potion screen through the screen's center
  - Light streaks from bright object can be simulated using steerable filter on a down sampled image



Lens flare (Battlefield)



Lens flare simulation <https://resources.mpi-inf.mpg.de/lensflareRendering/pdf/flare.pdf>

# Depth of field

- Depth of field: range where objects are in focus. Outside objects are blurred
- Smaller camera aperture size → increased depth of field
  - Pinhole cameras: infinite depth of field
- Various methods exist. One solution:
  - Create separate image layers: e.g. near, far
  - Near and far layers are blurred
  - Final image is made by combining blurred and layers in focus using back to front compositing



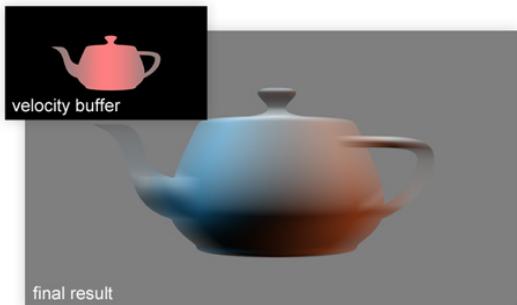
<https://www.versluis.com/2022/01/ue4-dof/>

<https://catlikecoding.com/unity/tutorials/advanced-rendering/depth-of-field/>

<https://developer.nvidia.com/gpugems/gpugems/part-iv-image-processing/chapter-23-depth-field-survey-techniques>

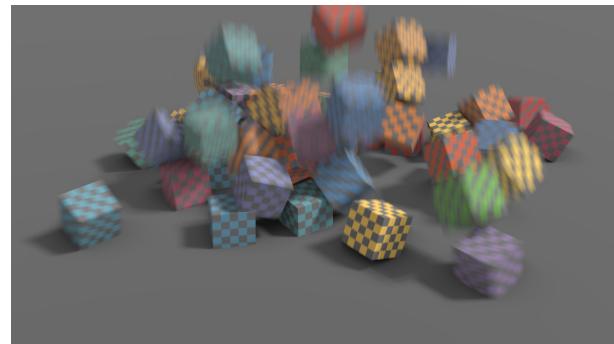
# Motion blur

- Motion blur comes from **movement of an object** across the screen or from **camera movement**
- Different sources of motion blur require different methods
  - Camera orientation changes → e.g., **circular blur** around view axis if camera is rotating around view axis or **directional blur** based on orientation direction
  - Camera position changes → **radial blur**
  - Object position/orientation changes → motion of each object is computed using **depth or velocity buffer** for determining amount of blur



Object rotation and position change blur using velocity buffer

<http://john-chapman-graphics.blogspot.com/2013/01/per-object-motion-blur.html>



Directional blur due to camera orientation change (Tomb raider)



Radial blur due to camera position change (Need for speed)

# Displaying and storing images

# Digital imagery

- Digital imagery appears in all forms of media. Most of these are:
  - Digital photographs or other types of 2D pictures scanned or loaded in computer
  - **Images generated using computer graphics: 3D modeling and rendering software**



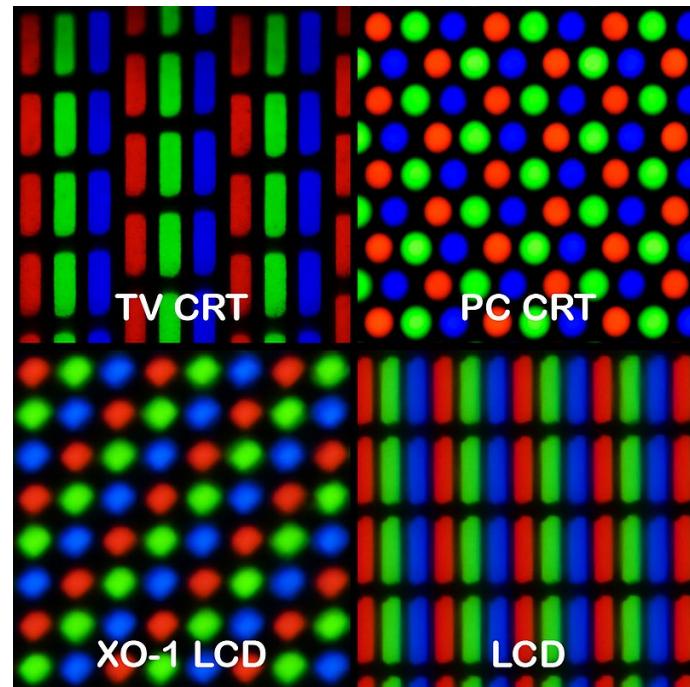
Digital photography <https://www.warnerbros.com/movies/blade-runner-2049>



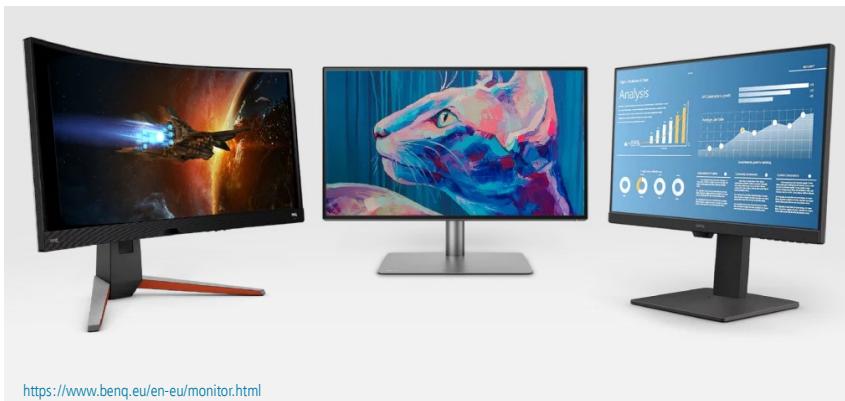
Rendered image: <https://www.thewitcher.com/en/witcher3>

# Display device

- Digital images are displayed on a **raster display device**  
→ **raster images**
  - e.g., monitor, television screen, etc.
- Raster display device is made of **arrays of pixels** → **discrete representation**
  - For example, each pixel of image created with digital camera (or rendering) stores red, green and blue value used to drive the red, green and blue values of screen pixel



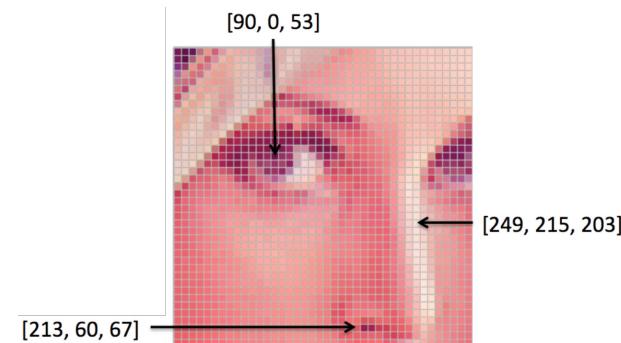
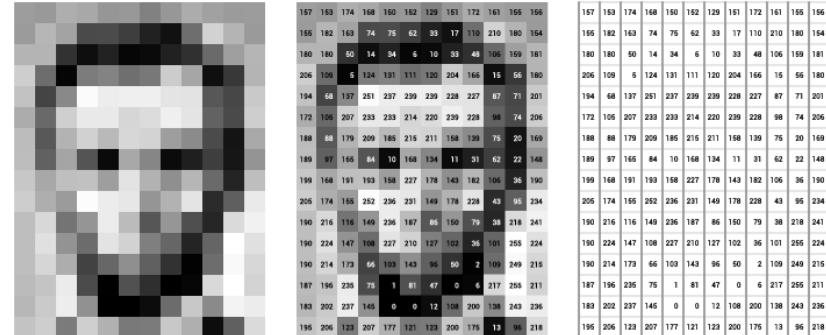
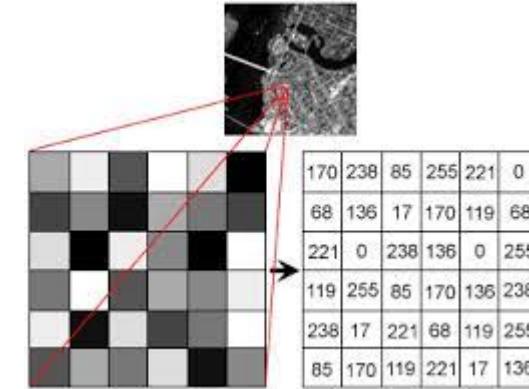
[https://en.wikipedia.org/wiki/Pixel\\_geometry](https://en.wikipedia.org/wiki/Pixel_geometry)



<https://www.benq.eu/en-eu/monitor.html>

# Raster image graphics

- Raster images are **rectangular array of values** called **pixels**.
- All values in such array have the same type, examples:
  - Floating point numbers representing levels of **gray**
  - Floating point triplets representing mixture of red, green and blue (**RGB**)
  - Floating point numbers representing **depth** from camera
- Such rectangular array of numbers can be interpreted in many ways
  - Arbitrary information can be stored/encoded in such array
  - Numbers in array do not have particular significance until stored in certain **standardized file format**



# Naive image file format: PPM

- Portable Pixel Map (PPM): text-based (P3) image file format

- Header:

- Width ( $w$ ) and height ( $h$ ) of image is specified
  - Maximum color value

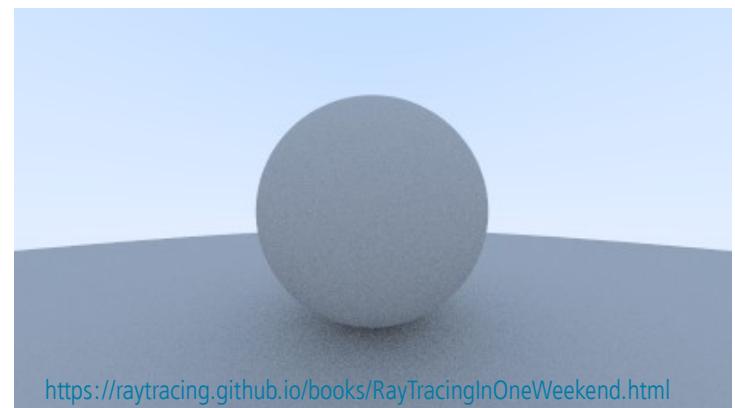
- Pixel values:

- $3 * w * h$  color values representing red, green and blue components of pixel
  - Pixel values ordering: left-to-right, top-to-bottom
  - Separated by white-space

- Naive image file format:

- Easy to write and analyze
  - Inefficient and highly redundant (compression is not present)
  - Little information about image

```
P3
# feep.ppm
4 4
15
 0  0  0   0  0  0   0  0  0   15  0 15
 0  0  0   0 15  7   0  0  0   0  0  0
 0  0  0   0  0  0   0 15  7   0  0  0
15  0 15   0  0  0   0  0  0   0  0  0
```



<https://raytracing.github.io/books/RayTracingInOneWeekend.html>

# Raster image file formats

Format	Channel Depth	Alpha	Meta data	DPI	Extensions
BMP	8bit	✗	✗	✓	.bmp
Iris	8, 16bit	✓	✗	✗	.sgi .rgb .bw
PNG	8, 16bit	✓	✓	✓	.png
JPEG	8bit	✗	✓	✓	.jpg .jpeg
JPEG 2000	8, 12, 16bit	✓	✗	✗	.jp2 .j2c
Targa	8bit	✓	✗	✗	.tga
Cineon & DPX	8, 10, 12, 16bit	✓	✗	✗	.cin .dpx
OpenEXR	float 16, 32bit	✓	✓	✓	.exr
Radiance HDR	float	✓	✗	✗	.hdr
TIFF	8, 16bit	✓	✗	✓	.tif .tiff
WebP	8bit	✓	✓	✓	.webp

# Raster image file formats

- Images are stored in many formats, closely related to the **display format**
- File formats use different **compression strategies**:
  - **Lossless** compression is one in which data occupies less space but from which original data can be reconstructed.
  - **Lossy** compression is resulting in even less occupied space, but original data can not be restored. Nevertheless, existing data is enough for intended use
- Content is described in terms of **channels**
  - e.g., red values for all pixels represents red **color channel**
  - e.g., depth values form **depth channel**
  - e.g., transparency values from **alpha channel**
- Some formats additionally store **metadata**
  - e.g., bit depth of color channel (e.g., 8 bit)
  - e.g. information on when and with which program the image was produced

# File formats in production

- Digital painting and image manipulation
  - Krita: bmp, jp2, **jpeg**, ora, pdf, **png**, ppm, raw, **tiff**, xcf <https://krita.org/en/item/krita-features/>
  - Gimp: XCF, **JPG**, **PNG**, GIF, **TIFF**, Raw, etc. <https://www.gimp.org/tutorials/ImageFormats/>
- 3D modeling
  - Blender: BMP, Iris, **PNG**, **JPEG**, Targa, OpenEXR , **TIFF**, HDR, etc.  
[https://docs.blender.org/manual/en/latest/files/media/image\\_formats.html](https://docs.blender.org/manual/en/latest/files/media/image_formats.html)
  - Houdini: **png**, gif, **jpg**, **tiff**, etc. [https://www.sidefx.com/docs/houdini/io/formats/image\\_formats.html](https://www.sidefx.com/docs/houdini/io/formats/image_formats.html)
- Game engines:
  - Unity: BMP, **TIF**, TGA, **JPG**, **PNG**, PSD, etc. <https://docs.unity3d.com/2019.2/Documentation/Manual/AssetTypes.html>
  - Godot: BMP, OpenEXR, **JPEG**, **PNG**, SVG, etc.  
[https://docs.godotengine.org/en/stable/tutorials/assets\\_pipeline/importing\\_images.html](https://docs.godotengine.org/en/stable/tutorials/assets_pipeline/importing_images.html)

# Raster image file formats

- Often used raster image file formats are:
  - Joint Photographic Experts Group - **JPEG**
  - Portable Network Graphics - **PNG**
  - Tag Image File Format - **TIFF/TIF**
- Advanced image file formats:
  - **OpenEXR** (<https://github.com/AcademySoftwareFoundation/openexr>)
  - **HDRI**: High Dynamic Range Image
- Library for handling different file formats for computer graphics applications:  
<https://github.com/OpenImageIO/oii>

# JPEG

- Recommended for display and storage of photography

## Pros:

- Efficient file **compression**
- **Universally** supported for display
- Most **digital cameras** produce JPEG images

## Cons:

- **Lossy**
  - Problematic for comparing the image due to differences in compression algorithm
  - Repeated editing degrades image quality
- **Artifacts** can be seen for computer generated graphics and text
- **Doesn't support transparency**
- Color channels are coded on 8 bits

# PNG

- Developed as an improved replacement for Graphics Interchange Format (GIF)
- Recommended for web page widgets, computer graphics and screenshots

## Pros:

- **Lossless** format
- Supports **transparency**
- **Small file size** for most computer graphics generated images
- Support by all **browsers**
- PNG file format is more compact than PPM and equally **easy to use**
- Support for **RGB(A)** and **grayscale images**

## Cons:

- Complex images are heavier
- Color channels coded in **8 bits**

# TIFF

- Used for storage and exchange of high quality images
- TIFF is ideal for representing intermediate or final results
  - In image editing and compositing tools, multiple images are often blended or laid atop one another

## Pros:

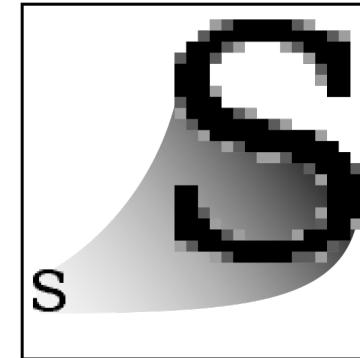
- Color coding in **16 bits**
- **Supported** by all image processing software
- TIFF images store **multiple channels** (images/layers)
  - Each channel store description of contents
- Various **compression** schemes
  - **uncompressed**
  - Compressed with **lossless** scheme
  - Compressed with **lossy** scheme

## Cons:

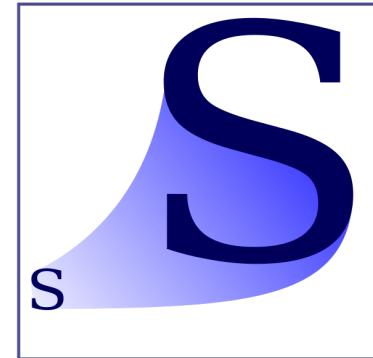
- **Heavier** on complex images

# Vector graphics images

- Alternative to raster graphics image is **vector graphics image**
  - Images are created directly from **geometric shapes defined on 2D Cartesian coordinate system**
  - Information in such image is **stored as points, lines, curves and polygons**
- Today, raster-based monitors and printers are typically used
  - Nevertheless, vector data and software is **used in applications where geometric precision is required: engineering, architecture, typography, etc.**
  - To display an image, **rendering** is performed to evaluate analytically defines shapes on raster display



**Raster**  
GIF, JPEG, PNG



**Vector**  
SVG

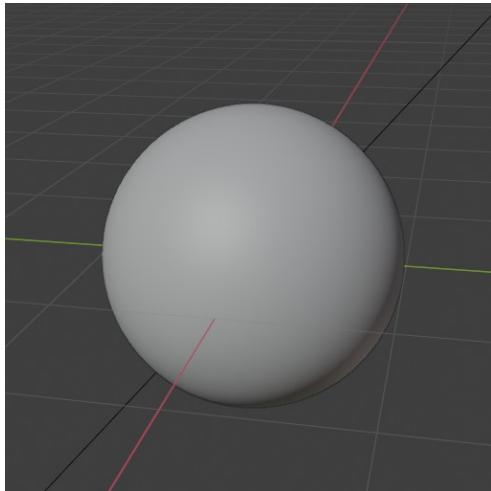
[https://en.wikipedia.org/wiki/Vector\\_graphics](https://en.wikipedia.org/wiki/Vector_graphics)

# Aliasing and anti-aliasing

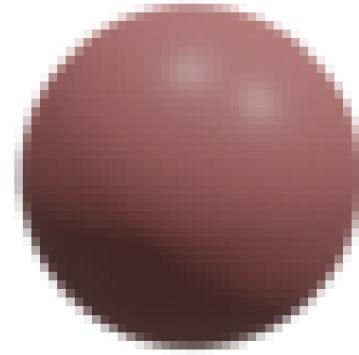
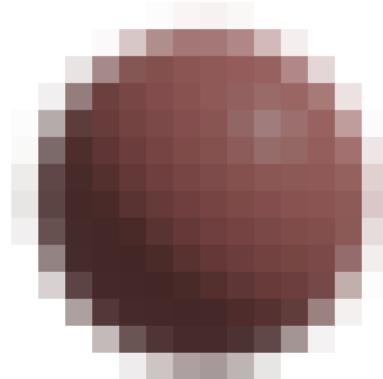
When continuous becomes discrete

# Continuous and discrete

- Images are **discrete** array of pixels
- 3D scene objects, lights and thus color values are **continuously** changing
- Rendering process, in order to create a discrete image of 3D scene, **samples the pixels on image plane**, finds corresponding sample in 3D scene and calculates color of the sample.

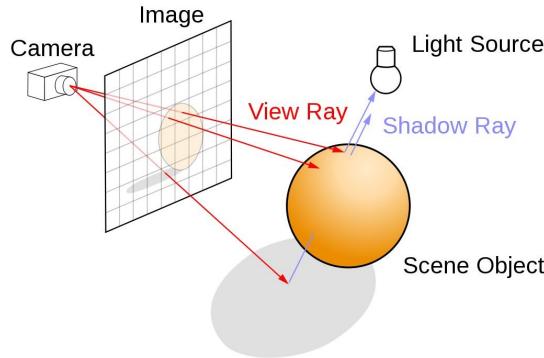


Continuous object in 3D scene



# Scene sampling and Image buffer

- Process of **rendering** images is inherently a **sampling task**
- Rendering is the process of **sampling a 3D scene** in order to obtain **colors for each pixel** in the image
  - Using camera information, **samples per image pixel** are generated
  - For each sample, **corresponding 3D scene point** is found and color is calculated (ray-tracing, rasterization)
  - Color of each pixel for each sample is combined into **image buffer**

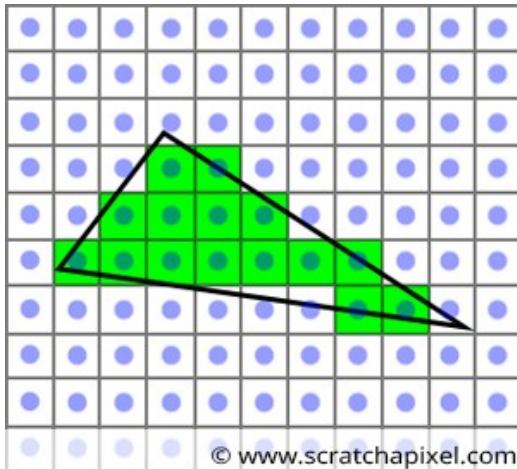


Note that pixel can have only one color!

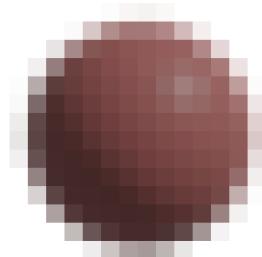
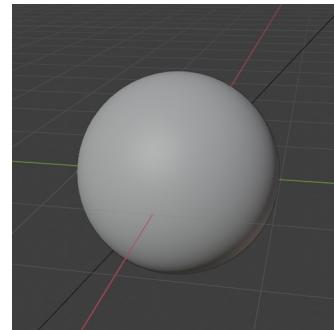
Pixels overlapping multiple objects/textures in 3D scene must be sampled with multiple rays to obtain correct representation.

# Aliasing

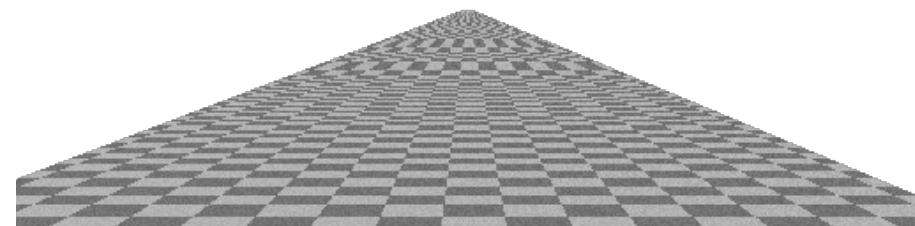
- Discretizing 3D scene (continuous information) may cause **Aliasing**
  - Jagged edges, flickering highlights (firelfys), Moiré pattern



Discretization



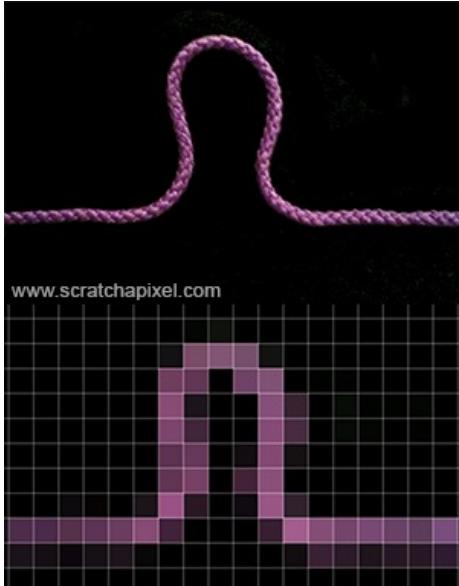
Problem: Jagged edges



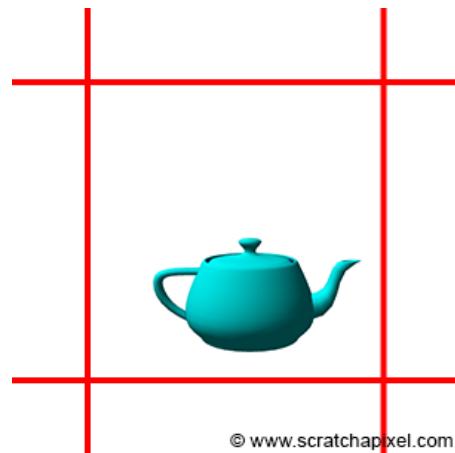
Problem: Moiré pattern

# Aliasing: intuition

- Projecting pixel into 3D scene can cover larger portion of 3D scene which contain multiple objects and thus color variation.
- Any kind of rapid (high-frequency) change in geometry edges, textures, shadows, highlights can cause aliasing.



Thread contains large amount of details covered by only one pixel



Extreme example: one whole object in pixel footprint.

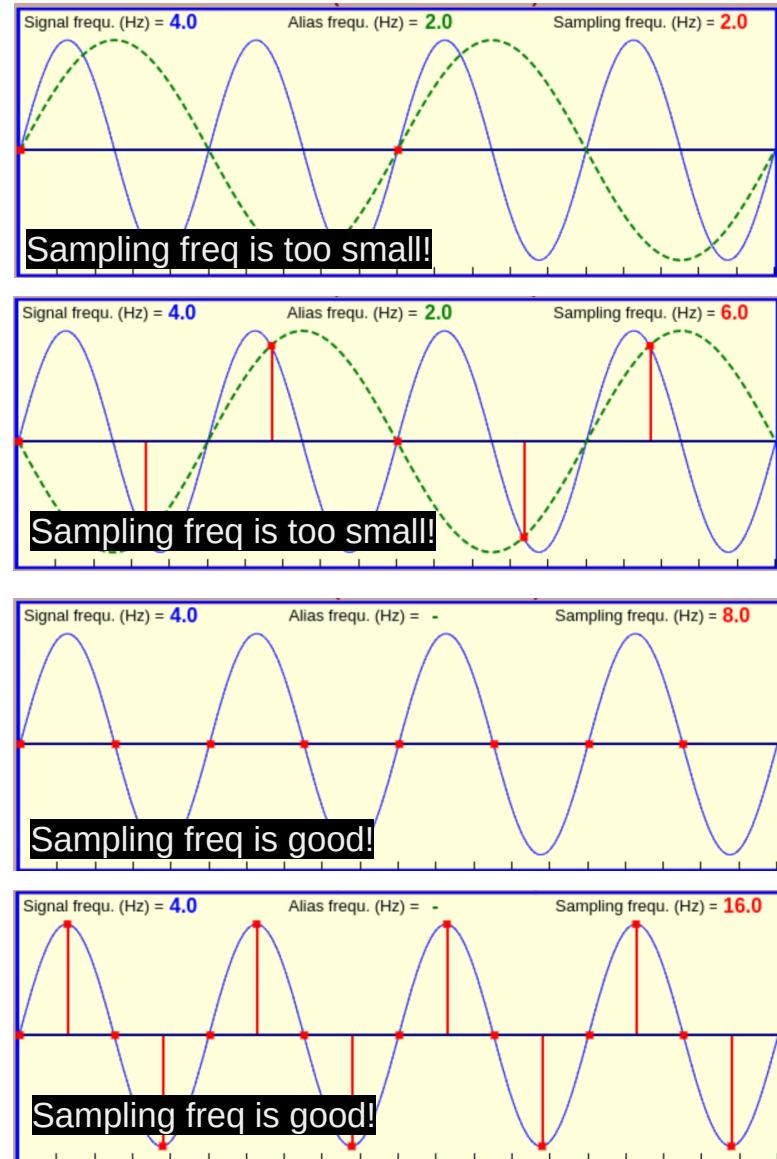


Changes in geometry, textures, highlights → sources of aliasing.

# Aliasing

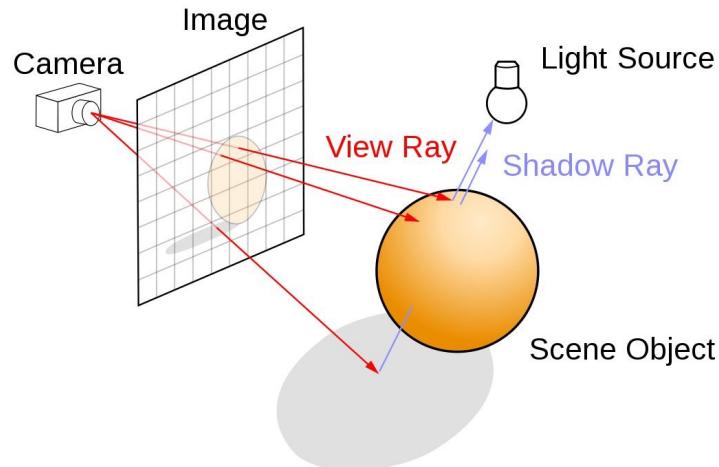
Blue: original signal  
Red: samples  
Green: reconstructed signal

- **Aliasing** – Occurs if signal is sampled at too low frequency
  - Reconstruction of signal from samples can not be performed correctly, e.g., reconstructing final image from pixel samples (**filtering**)
- **Sampling theorem: Sampling frequency must be twice the maximum frequency of sampled signal; Nyquist rate/limit**
  - As maximum frequency must be known for determining the sampling rate, **signal must be band-limited**
- **3D scenes are normally never band-limited** when rendering with point samples
  - **Sharp triangle or shadow edges** → discontinuous changes → infinite frequencies
  - **Rapid change of color**, e.g., specular highlights
- Point sampling is almost always used and entirely avoiding aliasing is not possible
  - Sampling pixel footprint which may contain lots of information requires a lot of samples



# Anti-aliasing

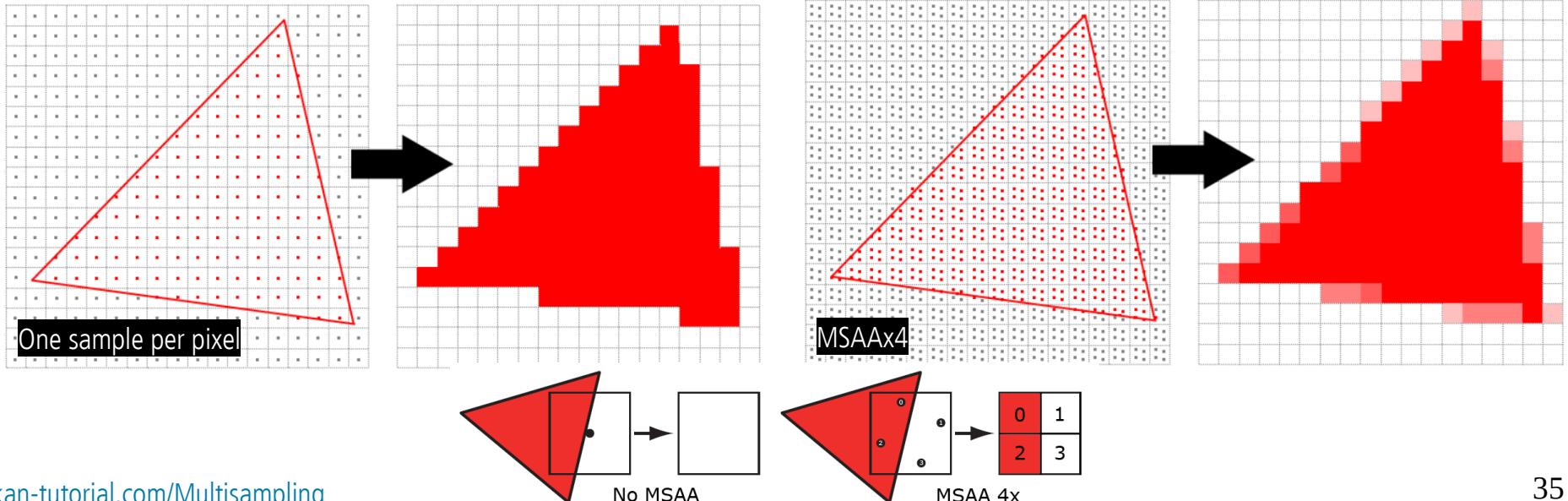
- Most common anti-aliasing method: **screen-based anti-aliasing**
  - Using multiple samples per pixel during rendering
- Anti-aliasing can be applied on: geometry, textures, shadows, highlights, etc.
  - At times, when it is possible to know if the signal is band limited and then anti-aliasing is performed
  - Trade-off between: quality, ability to capture sharp details or other phenomena, appearance, memory and speed



Instead of one ray per pixel, generate multiple rays - samples

# Multisample anti-aliasing

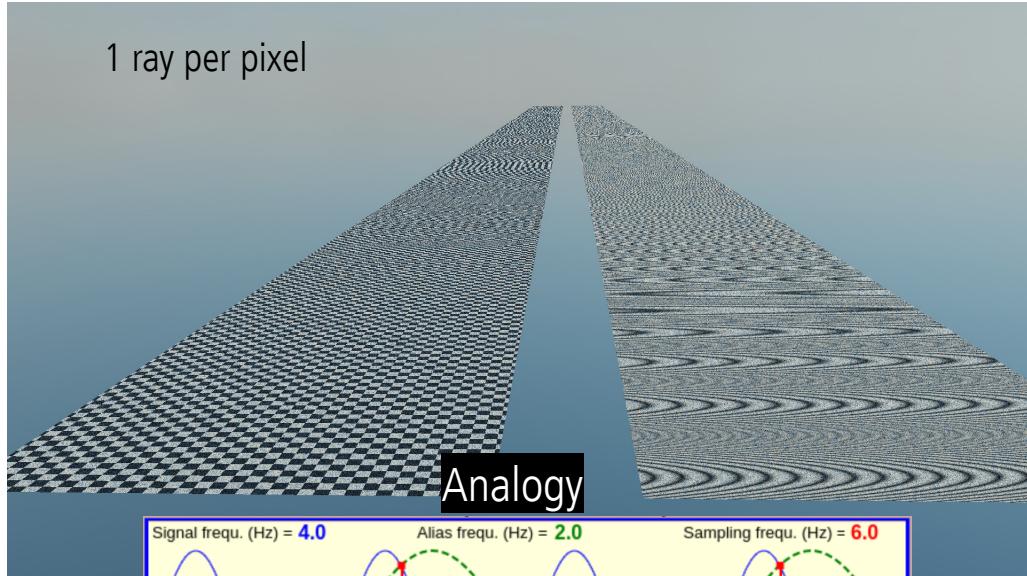
- Aliasing appears always when there are a lot of details under a pixel footprint in the scene
- To solve this, multiple sample per pixels can be used to “catch” high frequency details.
  - This method is called Multisample anti-aliasing (MSAA)



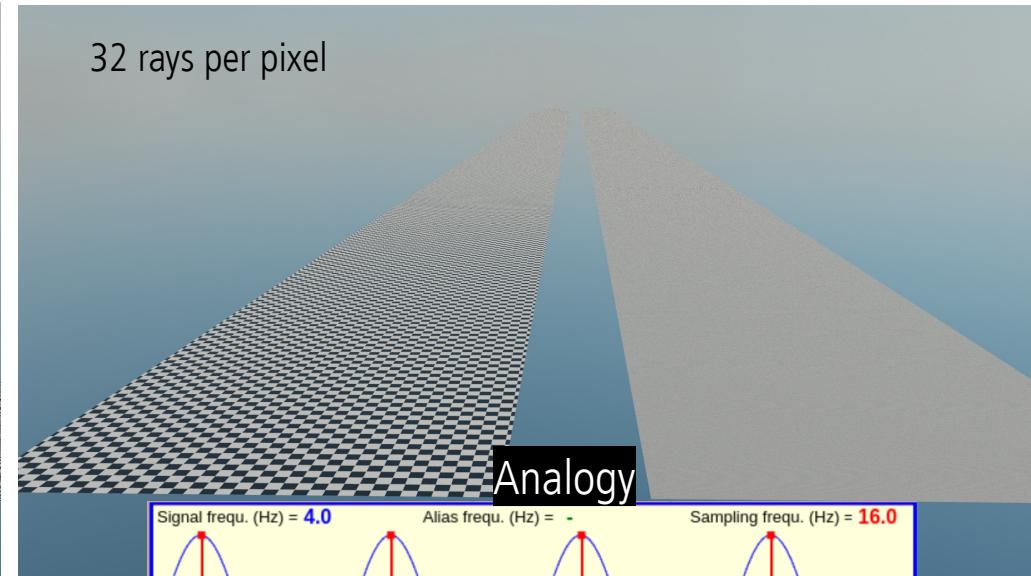
# Multisample anti-aliasing

- Using more samples per pixel reduces aliasing.
  - One sample per pixel is often not enough to reconstruct the image

1 ray per pixel

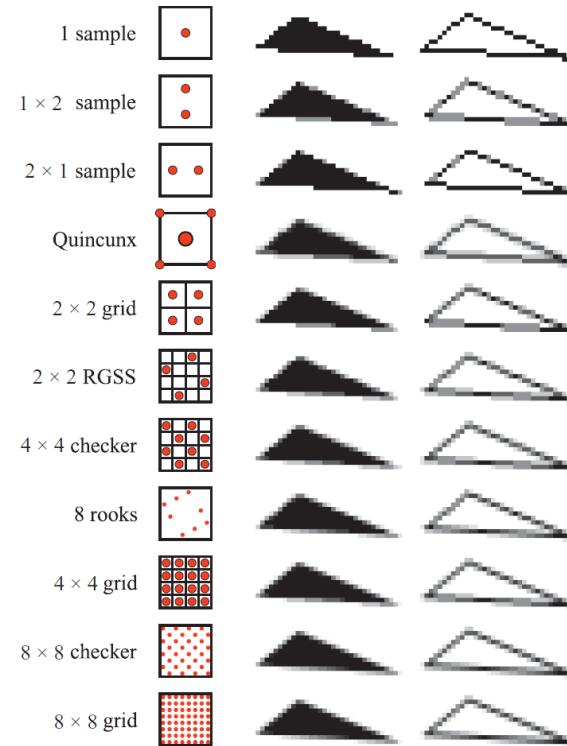


32 rays per pixel



# Antialiasing: sampling schemes

- Using more samples per pixel and combining those results in more representative pixel color
- Different pixel sampling schemes exists: trade-off between quality and speed



# Limitations of display device

# Limitation of display device

- **3D scene contains:**
  - Continuous description of objects (e.g., triangles, parametric surfaces, etc.)
  - Strong light sources and reflections (e.g., point lights, specular reflections)
- **Display devices are limited (discrete):**
  - Resolution (number of pixels)
  - Brightness (intensity)
  - Contrast
  - Color (gamut)
- **Rendered images** may contain values which can not be directly shown on display devices

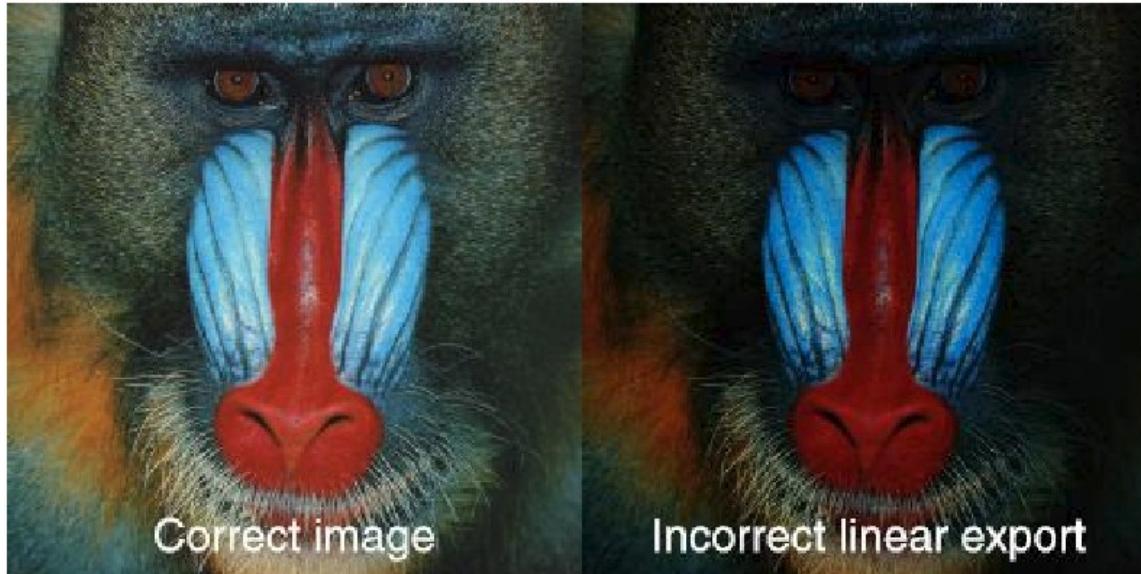
\* also optical-electrical transfer function describe the other end of the process for image and video capture devices

# Display limitations

- Display devices have **nonlinear relationship between input voltage and display amount of light**
  - As energy level applied to pixel is increased, the amount of light emitted doesn't grow linearly but according to power law → **nonlinear relationship**
    - e.g., pixel set to 50% will emit  $0.5^2$  amount of light
- Present since early display devices, e.g., cathode-ray tube (CRT) and also for modern display devices, e.g., LCDs
  - Relationship between digital color values in image buffer and amount of light emitted from the display is described with **electrical-optical transfer function (EOTF)\***

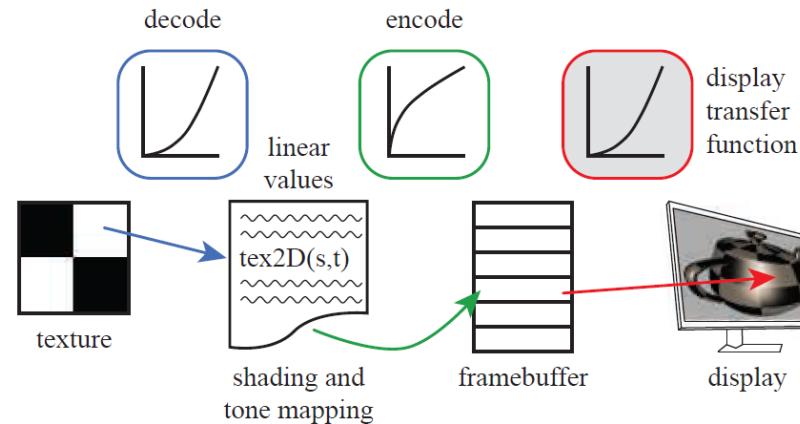
# Problem: displaying rendered images

- All input values (e.g., texture image) for rendering and all values during rendering computation must be in **linear** colorspace
  - Linear values are needed for correct addition and multiplication operations
- Rendered image containing **linear color space** will not display correctly if shown directly on a display device
  - Linear values will appear too dim on the screen



# Solution: display encoding

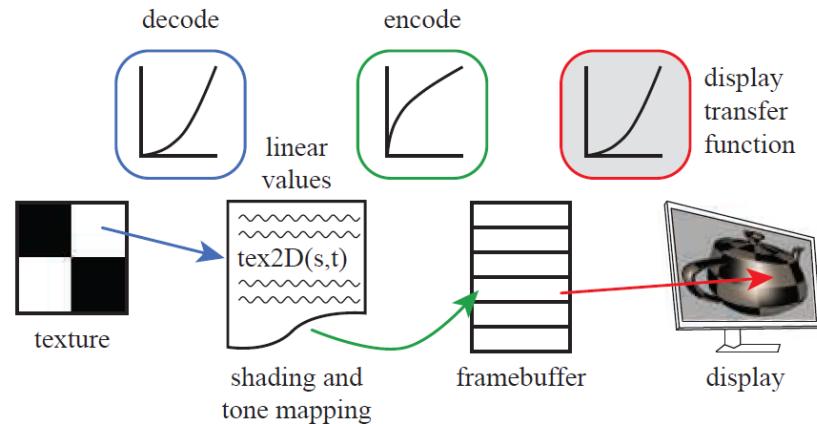
- When encoding linear color values for display, the goal is to cancel out the effect of display transfer function
  - Inverse of display transfer function (EOTF) is applied to color values in image buffer → **gamma correction**
  - Standard transfer function for personal computer display is called **sRGB**



# Gamma correction

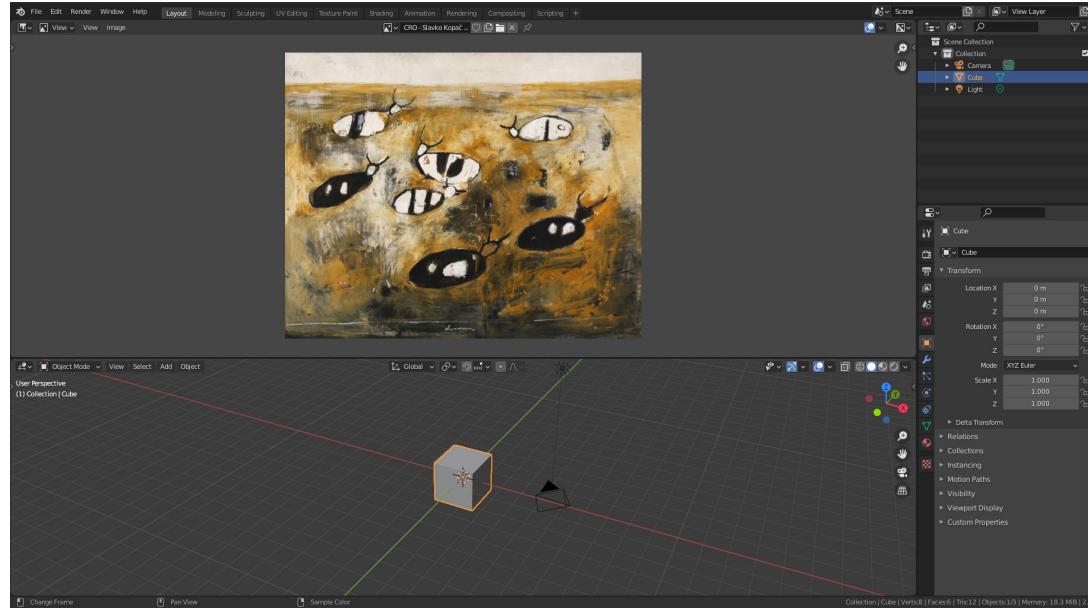
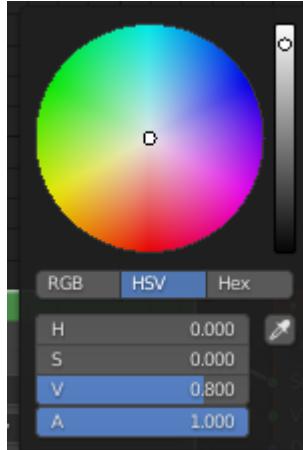
- Gamma correction is **last step in rendering and post-processing** – when everything is computed and image is ready for display.
  - To encode rendered image which is in linear colorspace ( $x$ ) for sRGB display ( $y$ ) the following formula is applied to all color values:

$$y = x^{1/\text{gamma}}, \text{ gamma} = 2.2$$



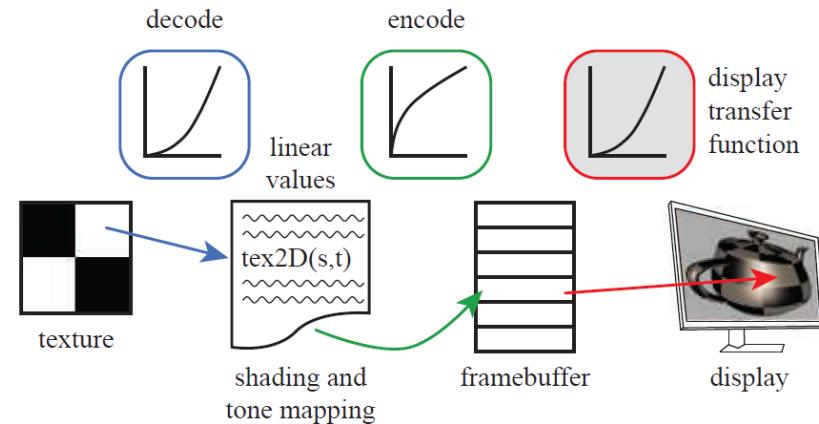
# Problem: preparing data for rendering

- When working in modeling tools users pick texture images or colors on screen - those colors are **encoded for display device** so we can see them properly
  - Those colors **can not be used in rendering computation directly** because they are in non-linear space



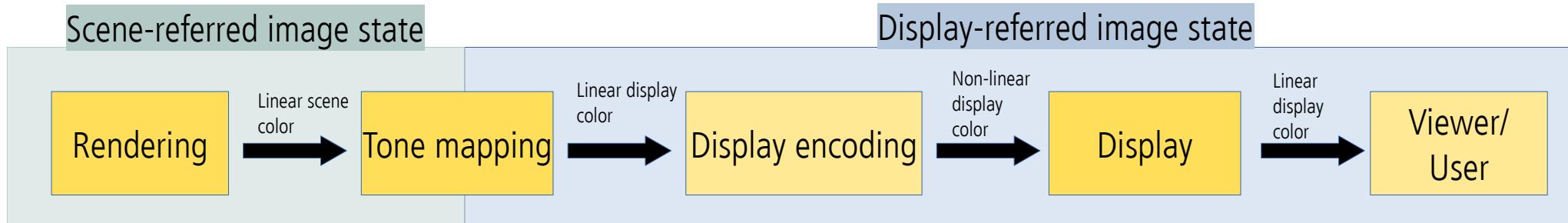
# Solution: inverse gamma correction

- Everything we see on the screen (e.g., image textures or color-pickers in modeling tools) is display-encoded data and those values **must be decoded to linear values for rendering**
  - To decode sRGB display encoded values ( $y$ ) to linear colorspace ( $x$ ) the following formula is applied to all color values:
    - $x = y^{\text{gamma}}$ , where  $\text{gamma} = 2.2$



# Imaging pipeline

- **Imaging pipeline** describes color handling from initial rendering to final display.
- **Rendering:** calculating image color based on 3D scene description.
  - The results are pixel values in color buffer – display results still needs to be determined.
- **Display encoding** is process in which linear color (radiance) values are converted to nonlinear values for display
- **Tone mapping:** performed between rendering and display encoding



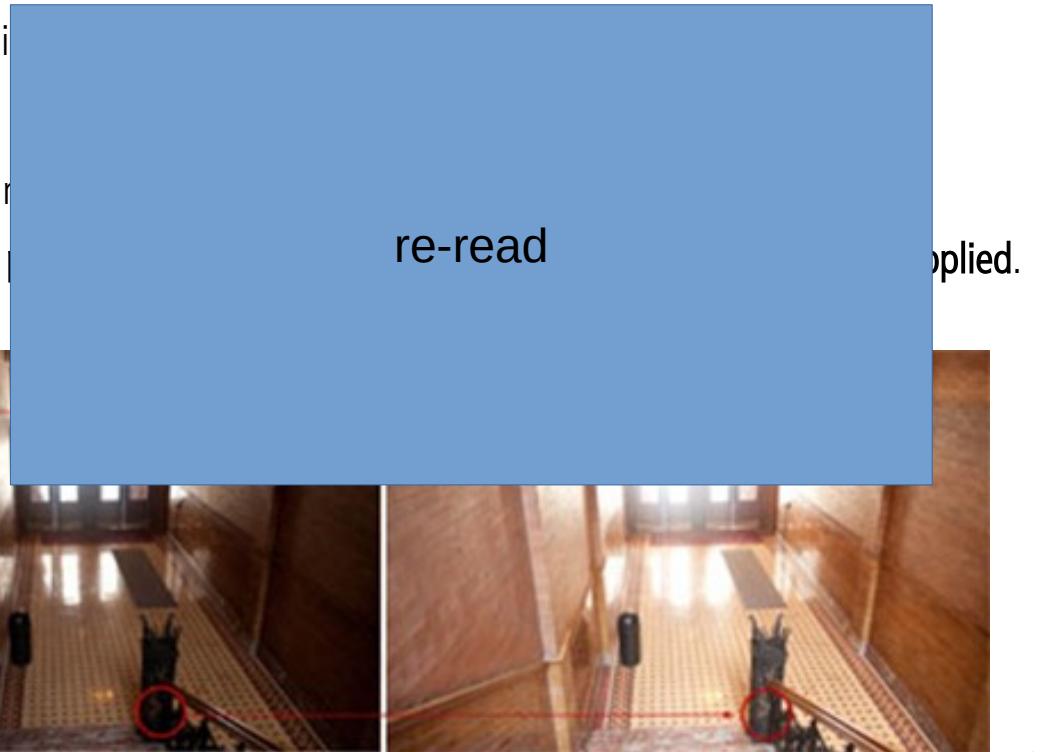
- Defined in reference to scene color (radiance) values
  - Physically-based rendering computations
- Defined in reference to display color (radiance) values.
  - Appropriate for viewing

# Tone mapping

- Tone mapping (end-to-end or scene-to-screen transfer) is about converting scene color values to display color (radiance) values
  - Goal 1: **image reproduction** - create display-referred image that reproduces, as closely as possible given display and viewing properties, perceptual impression that viewer would have if they were observing original scene.
  - Goal 2: **preferred image reproduction** – create display-referred image that looks better (to some criteria) than original scene.

# Tone mapping: (1) image reproduction

- Reproduce a similar perceptual impression as the original scene, problems:
  - Original scene **luminance and saturation** exceeds display range
- **Exposure** is critical to image reproduction
  - Exposure in photography refers to controlling the amount of light applied.
  - In rendering, exposure is a **linear scaling operation**



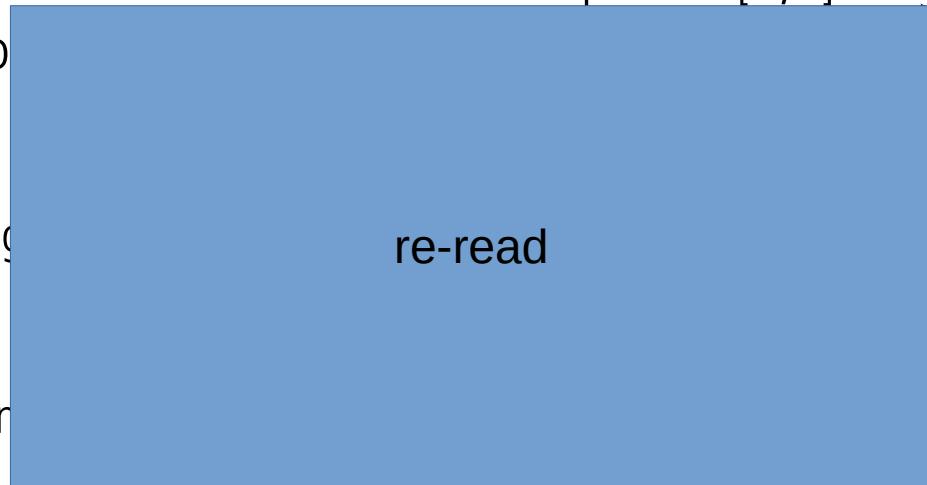
# Image reproduction: exposure

Different exposures of the same render:  
[https://docs.blender.org/manual/en/latest/render/color\\_management.html](https://docs.blender.org/manual/en/latest/render/color_management.html)



# Image reproduction

- Process of transforming floating point scene color values to the expected [0, 1] range for low dynamic range display without loss of quality
- **Global tone mapping:**
  - Same mapping applied to the whole image
- **Local tone mapping:**
  - Different mapping pixel-to-pixel based on neighborhood



# Global tone mapping: tone reproduction transform

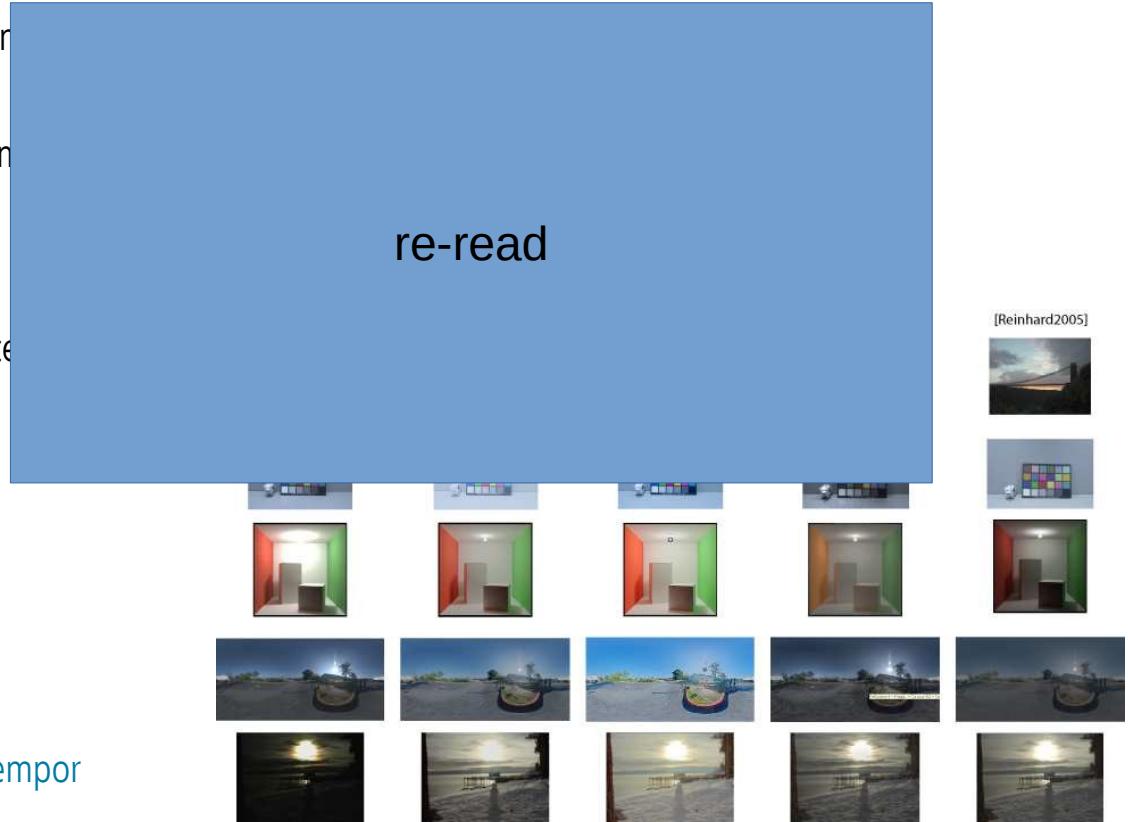
- Reinhart tone reproduction operator

- Commonly used method for computing exposure: contrast of pixel brightness of rendered image
- Leaves dark values unchanged and bright values asymptotic to white

- ACES tone mapping

- Standard for tone managing for motion picture and television
- Emerging in real-time applications: Unity and Unreal Engine

- Other tone mapping transforms



# Tone mapping: (2) preferred image reproduction

- **Color grading:** creative manipulation of image colors to obtain image desired artistic look
- **Grading look-up tables (LUTs)** contain desired color transformations which are applied on image
  - e.g., baked color curves
- Color grading is possible on:
  - Scene-referred images: produce high fidelity results
  - Display-referred images: easier to set-up

# Color grading



# More into topic

- Image space effects:
  - Motion blur: [https://www.nvidia.com/docs/io/8230/gdc2003\\_openglshaderticks.pdf](https://www.nvidia.com/docs/io/8230/gdc2003_openglshaderticks.pdf)
  - Motion blur: <https://developer.nvidia.com/gpugems/gpugems3/part-iv-image-effects/chapter-27-motion-blur-post-processing-effect>
  - Motion blur: <https://www.bryancphail.com/wp/?p=600>
  - Motion blur: [https://docs.blender.org/manual/en/latest/render/cycles/render\\_settings/motion\\_blur.html](https://docs.blender.org/manual/en/latest/render/cycles/render_settings/motion_blur.html)
  - NPR: [https://en.wikipedia.org/wiki/Non-photorealistic\\_rendering](https://en.wikipedia.org/wiki/Non-photorealistic_rendering)
  - General post-processing: <https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@7.1/manual/integration-with-post-processing.html>
- Color:
  - <https://developer.nvidia.com/sites/default/files/akamai/gameworks/hdr/UHDColorForGames.pdf>
  - <https://cinematiccolor.org/>
- Color grading:
  - <http://filmicworlds.com/blog/minimal-color-grading-tools/>
  - <https://docs.unrealengine.com/4.27/en-US/RenderingAndGraphics/PostProcessEffects/ColorGrading/>

# Repository

- <https://github.com/lorentzo/IntroductionToComputerGraphics>