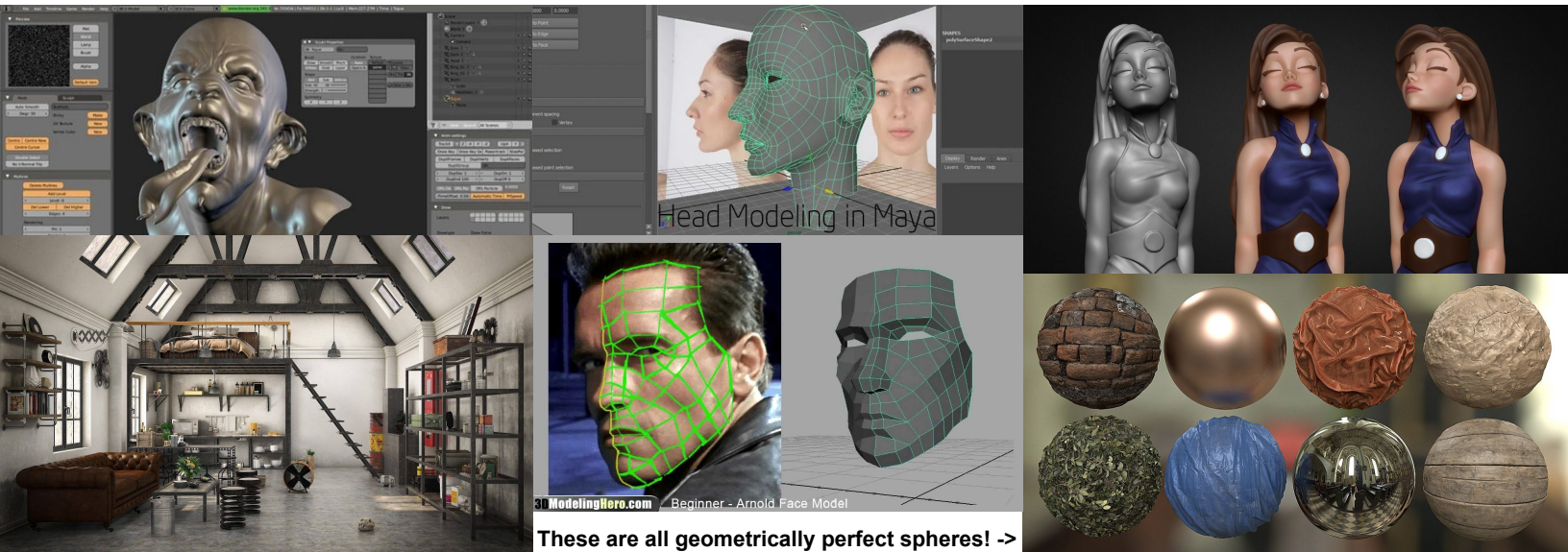


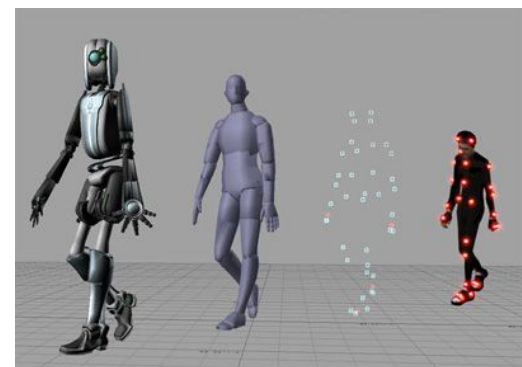
# High-Level Introduction to 3D Graphics



Images taken from Google images and sources are credited, when available. They are not copyrighted by the UMD Instructional Team or CS Department.

## “High-level”?

- **CMSC 427 – Introduction to Computer Graphics**
  - Hardware to software rendering pipelines
- **Simplifying the concepts**
  - Game engines **automatically** handle the low-level implementation
  - Game engines are basically **wrappers** for graphics APIs
- This lecture is an overview of CG pipeline



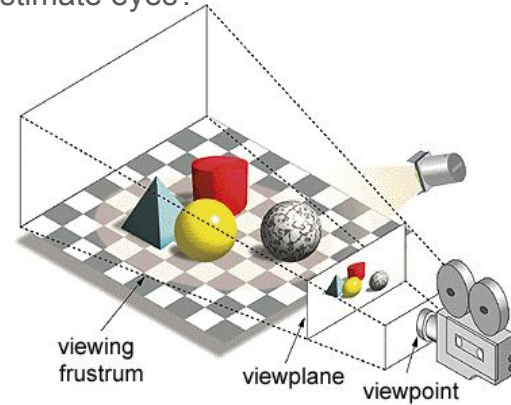
```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
// Reset transformations
glLoadIdentity()
// Rotate when user changes rotate_x and rotate_y
glRotatef(rotate_x, 1.0, 0.0, 0.0)
glRotatef(rotate_y, 0.0, 1.0, 0.0)
//Multi-colored side - FRONT
glBegin(GL_POLYGON)
glColor3f( 1.0, 0.0, 0.0)   glVertex3f( 0.5, -0.5, -0.5) # P1 is red
glColor3f( 0.0, 1.0, 0.0)   glVertex3f( 0.5, 0.5, -0.5) # P2 is green
glColor3f( 0.0, 0.0, 1.0)   glVertex3f(-0.5, 0.5, -0.5) # P3 is blue
glColor3f( 1.0, 0.0, 1.0)   glVertex3f(-0.5, -0.5, -0.5) # P4 is purple
```

# A fundamental problem: 3D>2D

- End result is almost always 2D
  - computer monitor, VR device screen, etc.
  - Exceptions: **holograms** (intersection of light rays create “3D” pixels)
    - E.g. 3D holographic projector, (some) autostereoscopic displays [1,2]
- Challenges:
  - **How to accurately project to 2D?** Distortion, visual effects, etc.?
  - **How to convince user that they’re looking at something 3D?** Estimate eyes?
- Optimizations:
  - How to clean the image?
  - How to make the pipeline efficient?
  - How to make the image photorealistic?
- Benefits:
  - Can create images very quickly if done well
  - Can make things look nice with visual trickery



From Computer Desktop Encyclopedia  
Reprinted with permission.  
© 1998 Intergraph Computer Systems

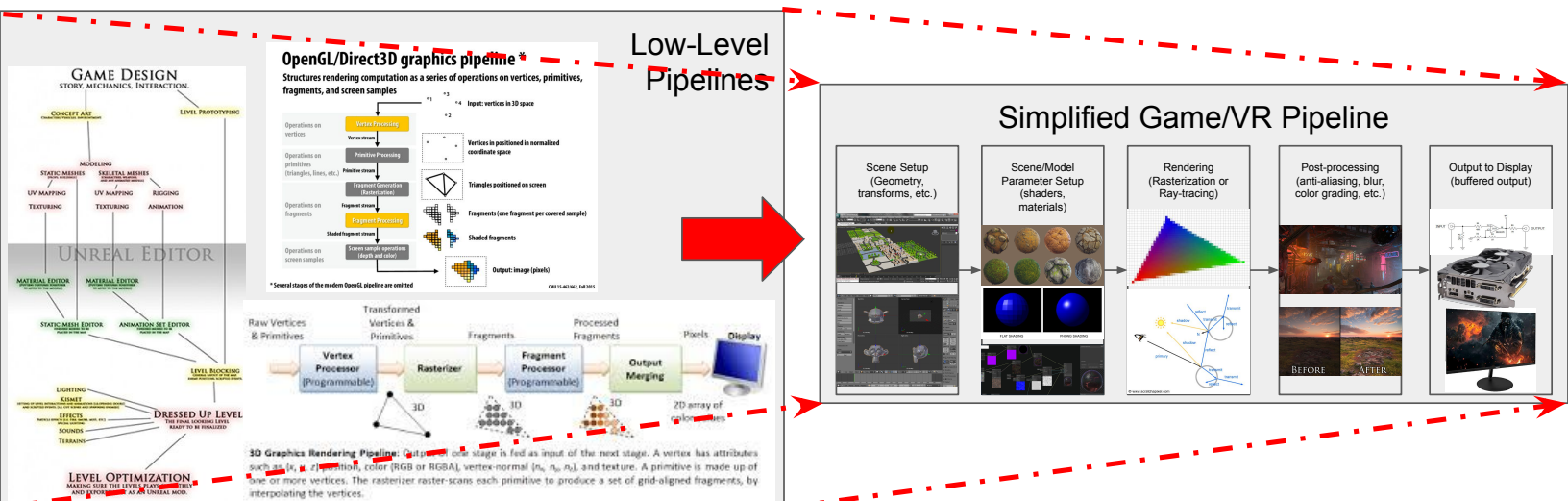


[1] Dodgson, Neil A. "Autostereoscopic 3D displays." *Computer* 38.8 (2005): 31-36.

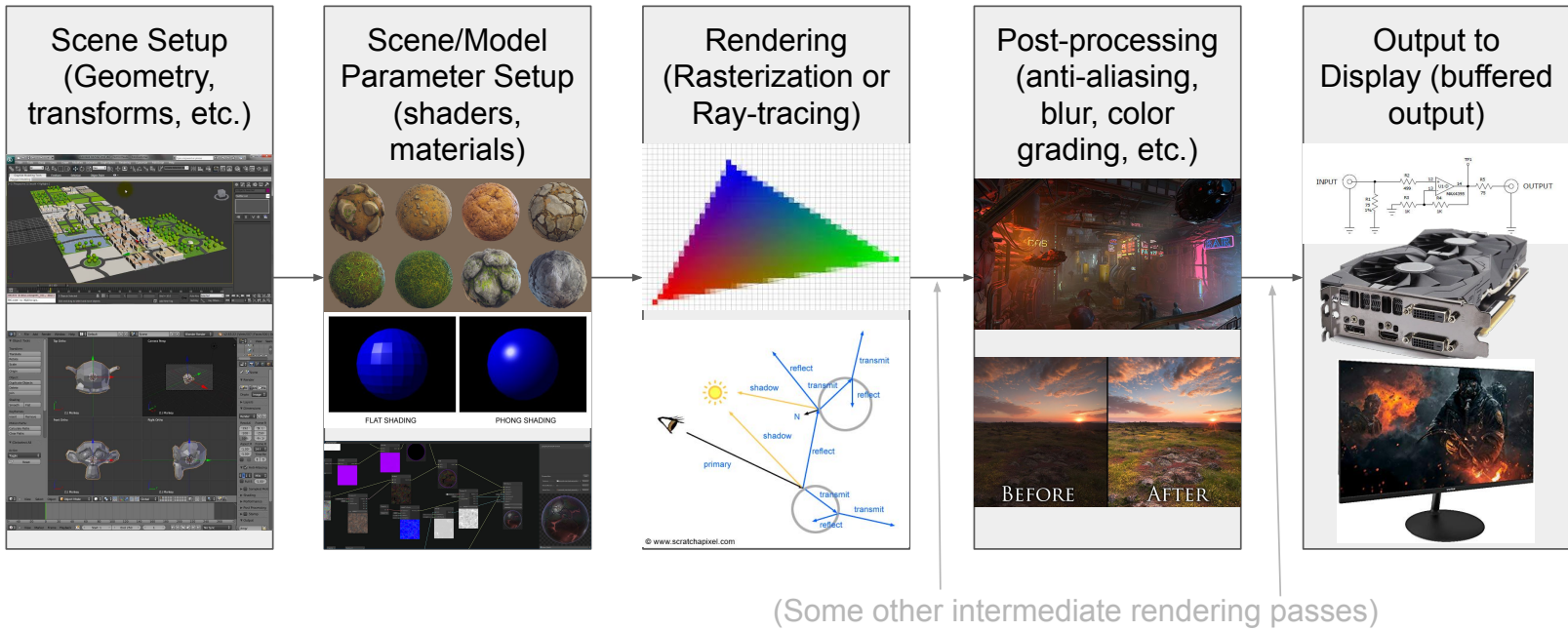
[2] Dodgson, Neil A., J. R. Moore, and S. R. Lang. "Multi-view autostereoscopic 3D display." *International Broadcasting Convention*. Vol. 2. 1999.

# High-Level 3D Graphics Pipeline

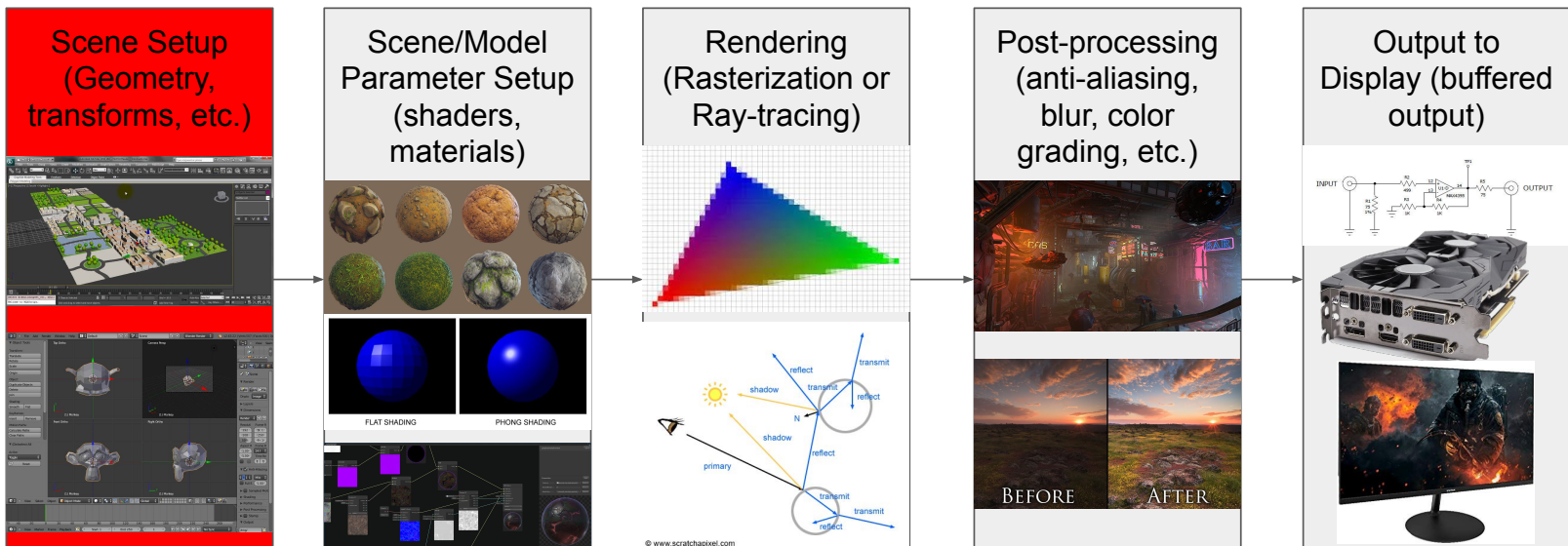
- Simplification of resources like OpenGL’s & UE3 pipeline documentation
- Pipeline from perspective of game/VR dev.... **Not exactly how it flows internally**
  - Think of it as “**order of things to worry about as a game/XR dev**”
  - Mix of graphics and development pipelines



# Simplified Game/XR Pipeline

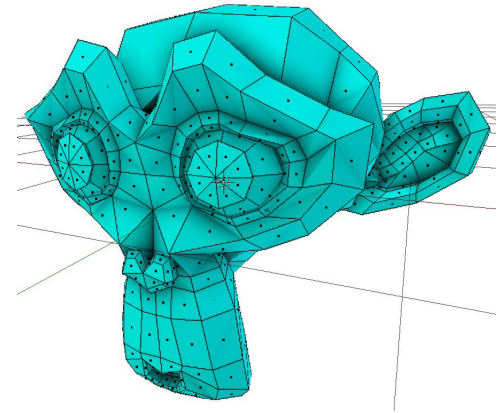
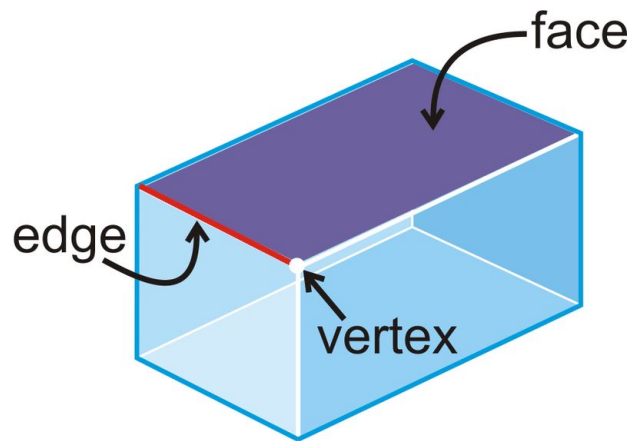
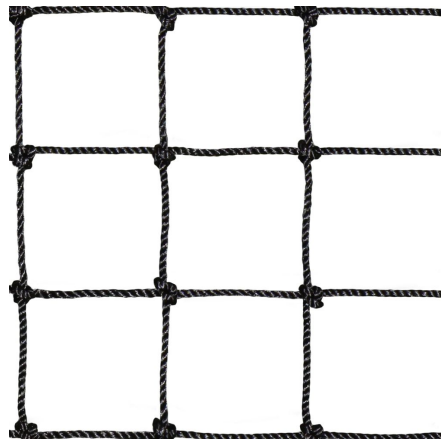


## Scene Setup



# 3D Models/Meshes

- **Vertices, edges, faces** (aka polygons or polys)
  - (usually tris/quads, game engines internally **triangulate** for optimizations and consistency)
- Often called **meshes**...bunch of vertices meshed together



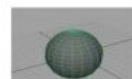
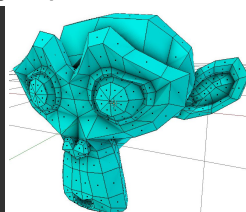
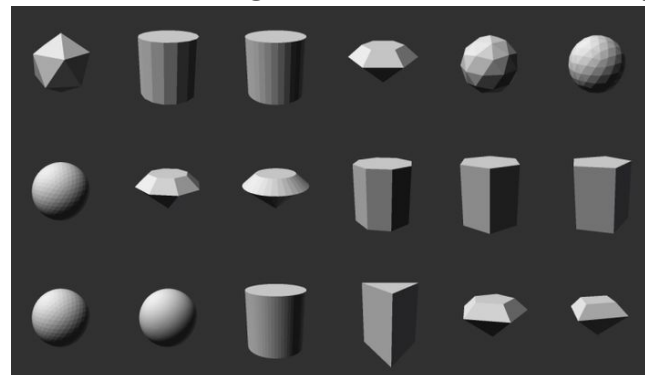
# 3D Primitives

- “**Atomic**” shapes: any mesh can be decomposed into **geometric primitives**
- Among the most important concepts in optimization
  - Extremely important for physics & collisions!
  - Also have rendering optimizations... not important for now
- Traditionally, primitives are **simple, convex shapes**
  - so no “dents”....topology doesn’t suddenly flip direction
- Definition changed over time to just mean **common shapes**
  - E.g. Blender considers monkey a primitive

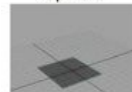


Concave

Convex



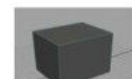
Sphere



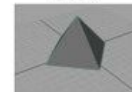
Plane



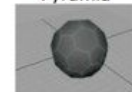
Prism



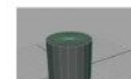
Cube



Pyramid



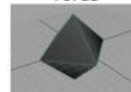
Soccer Ball



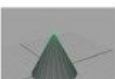
Cylinder



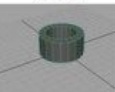
Torus



Platonic Solids



Cone



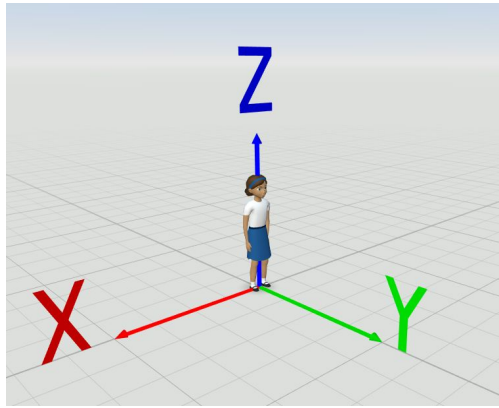
Pipe




Helix

# 3D Virtual Environments (VEs)

- aka scenes, levels, maps
- world/global origin (like the origin in any 3D axes)
- All things with physical definition have a **transform** (location/position, rotation/orientation, scale)
  - E.g. a class representing “game settings” doesn’t need transform
- **Global/world** transform relative to world origin
- **Local/relative** transform relative to a parent (e.g. want human eyes parented to the body)





TestScene

▼ Transform

Location:

X: 0.00000

Y: 0.00000

Z: 0.00000

Rotation:

X: 0°

Y: 0°

Z: 0°


XYZ Euler

Scale:

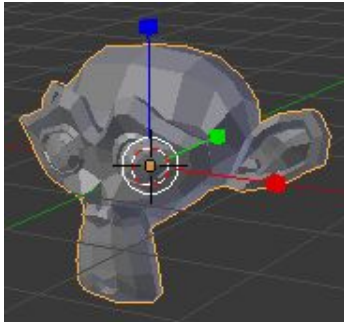
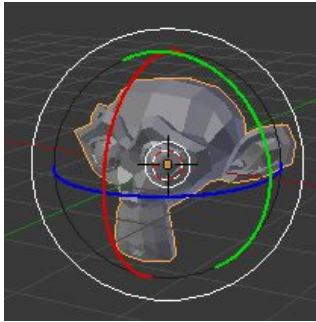
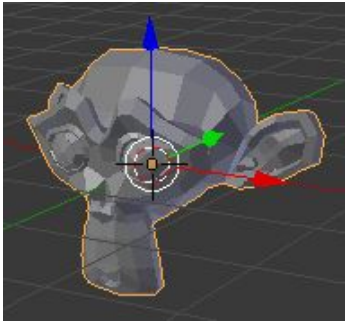
X: 1.000

Y: 1.000

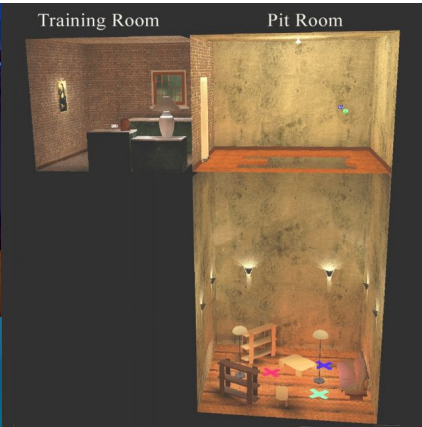
Z: 1.000



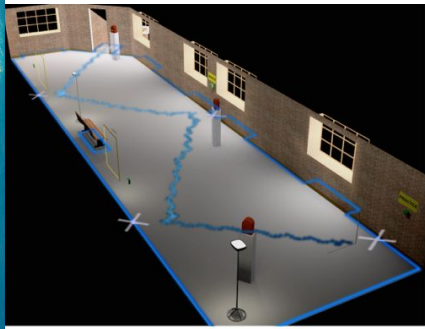
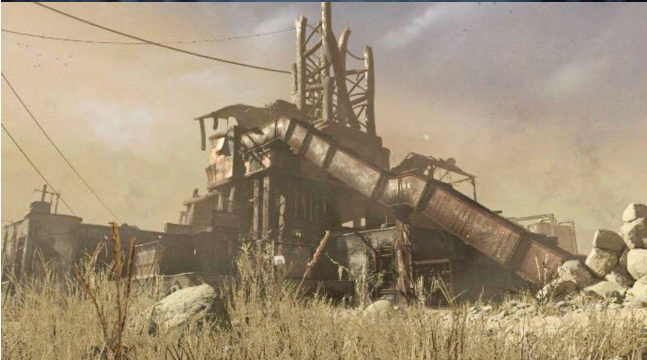
Digital Human



## More VEs



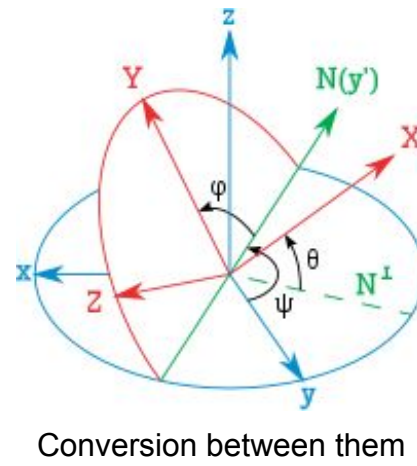
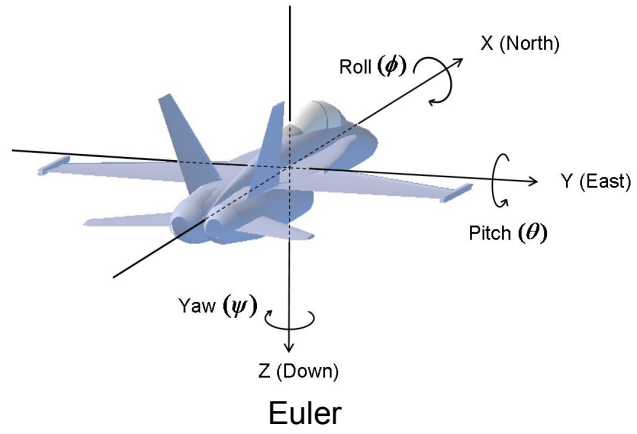
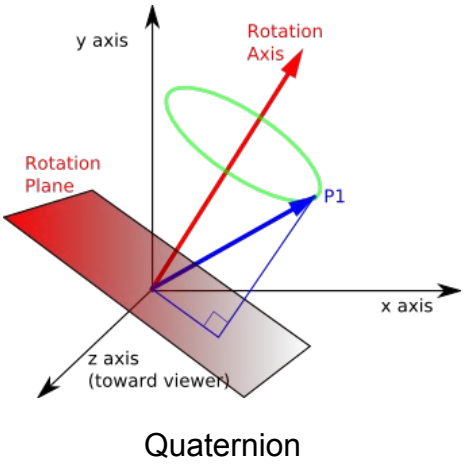
From Meehan 2002



From Razzaque 2001

# Rotations

- 2 standards:
  - **Quaternion:** composition of vectors: **(W, X, Y, Z)** [vector pointing forward & rotation around it]
    - Used more often in low-level graphics b/c they're easier to use in transformation matrices (which are usually 4x4)....which PCs are really good at computing
  - **Euler: (pitch, yaw, roll)**
    - Used in high-level APIs like game engines...although Quaternions usually used internally

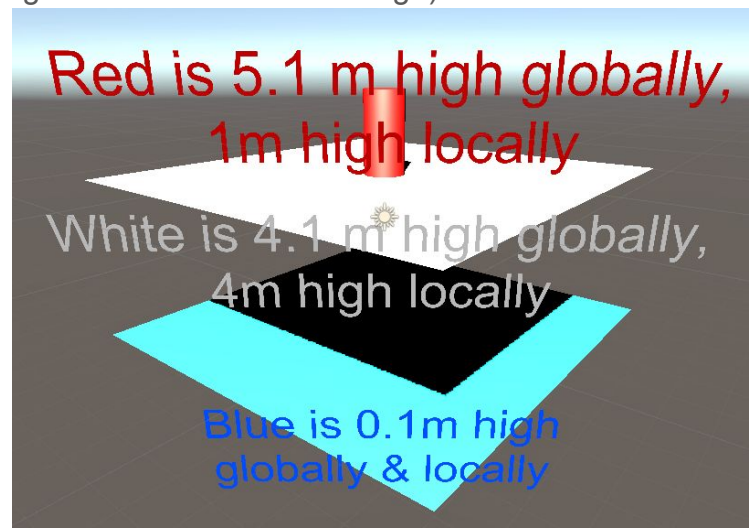


## From Local/Relative Space to Global/World Space

- Put very simply, if 3D model is the node of a tree:
  - traverse upwards through tree, adding all relative location & rotation, multiply scales
  - Stop after reaching world origin (aka root)
    - (transform of root relative to itself is [position=(0,0,0), rotation=(0,0,0), scale=(1,1,1)]....so you can keep iterating but the result won't change)

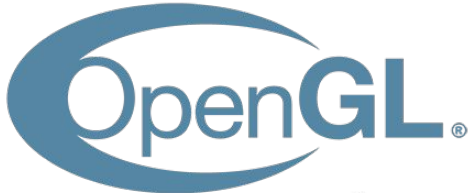


<-Name of Level is World Origin  
 Same convention in UE4  
 V

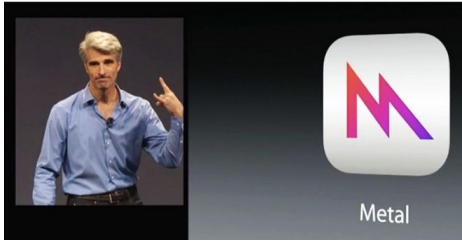


# Quick Intro to Low-Level Graphics APIs

- OpenGL (1992)
  - Made it possible to create graphics without going into hardware
  - Standardized graphics APIs
  - Still one of most widely compatible graphics APIs
  - Used by Unity and usually for simpler graphics
- Direct3D (1995) -> DirectX
  - DirectX describes entire range of MS's "Direct" APIs
  - Originally a competitor to OpenGL
  - Everyone petitioned to Microsoft to play nice
  - They did, but the APIs never merged as industry hoped
  - Now used by UE4 and higher-end graphics
- AMD Mantle (2013) -> Vulkan (2016)
  - Meant to balance CPU & GPU usage
  - Much lower-level
- Apple Metal (2015)
  - Poor attempt to disrupt the game engine industry (depracate OpenGL)



Microsoft®  
DirectX®

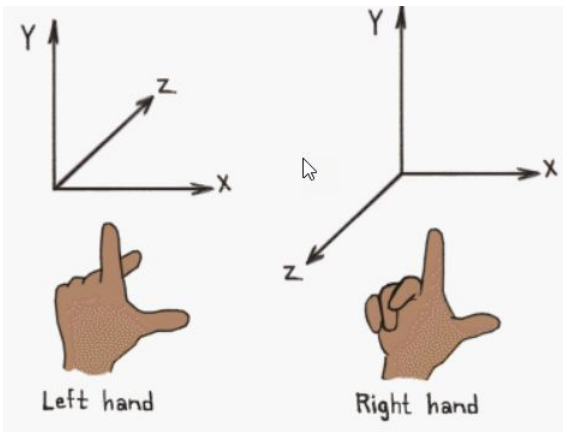


## Office Hours – to be updated as needs change

- **Open Lab Hours:** Mon-Thur 3:30pm-6pm; Fri 12:30pm -2:30pm  
@ AR/VR Instruction Lab (AVW 4120) by Suchismit Ghosh & Saunak Roy
- Office hours
  - [Ming C Lin](#) TuThur after class @ IRB 5162 or by appointment <[lin@umd.edu](mailto:lin@umd.edu)>
  - [Logan Stevens](#) TuThu 2-3pm or by request <[lsteven7@umd.edu](mailto:lsteven7@umd.edu)>
  - [Xiyi Chen](#) MW 3-4pm or by request <[xiyichen@umd.edu](mailto:xiyichen@umd.edu)>

# Coordinate Systems

- OpenGL-based systems (e.g. Unity) usually **Y-up**
  - Philosophy that XY plane is the screen and Z is **out of screen (depth)**... Physics does this
- DirectX-based systems (but not DirectX itself) (e.g. Unreal) usually **Z-up**
  - Philosophy that Z is **height**...which goes up in 3-space. Also follows 3D math conventions
- Lower-level graphics APIs (non-game engines) are usually right-handed
- Game engines (Unity, Unreal) usually left-handed
  - Forward vector, right vector, up vector are positive



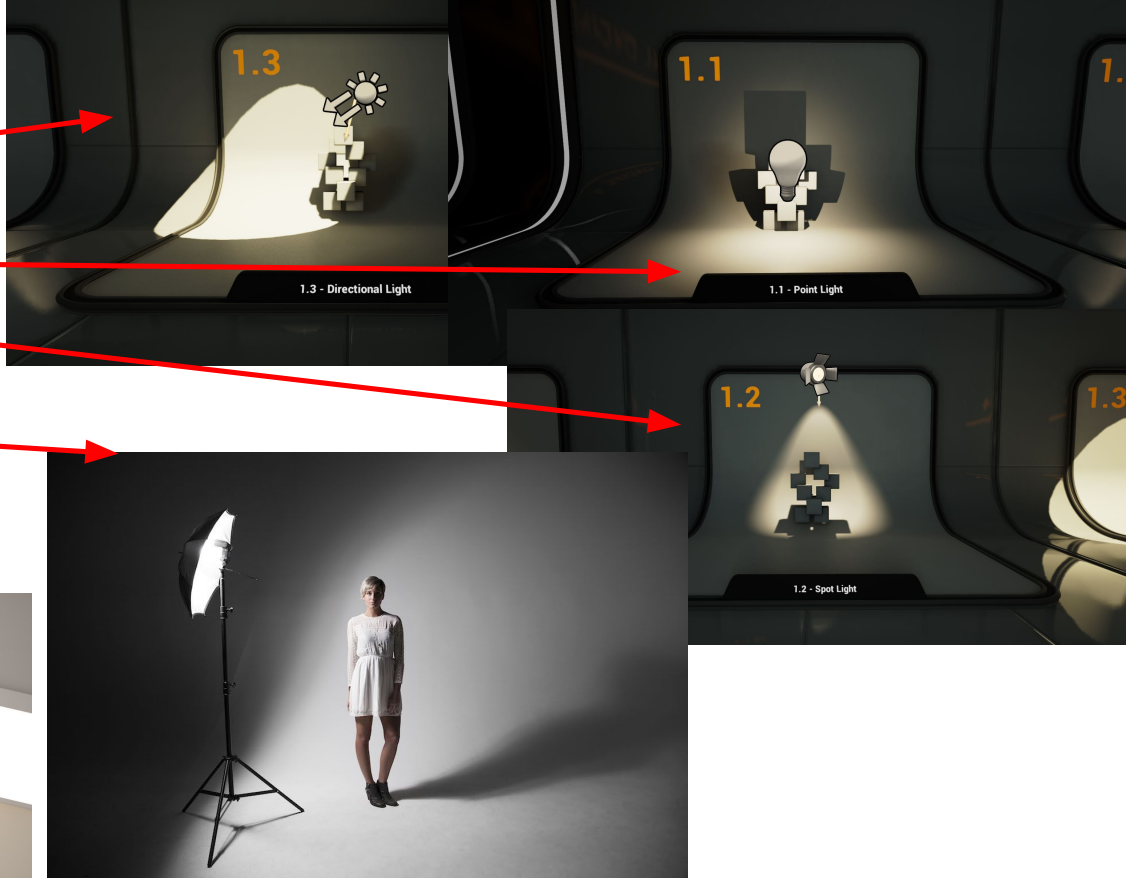
Standard Color scheme:  
**X**  
**Y**  
**Z**

(I use finger guns)

	Left-handed	Right-handed
Y is up	unity DirectX	MAYA OpenGL
Z is up	UNREAL ENGINE	source CRYENGINE blender

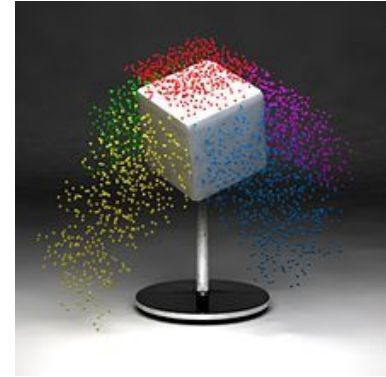
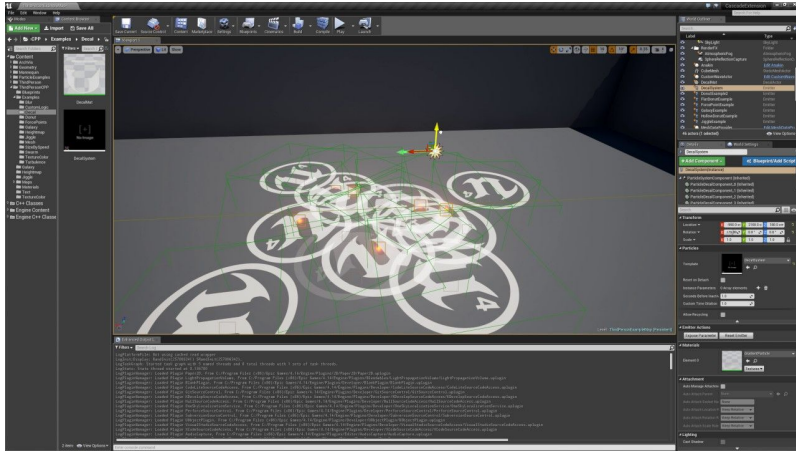
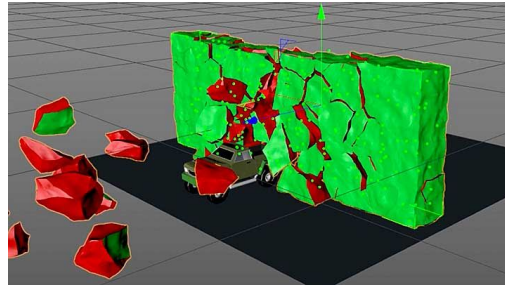
# Light Sources

- Directional
  - Used for sun
- Point
- Spot
- Ambient/SkyLight
- Planar
  - (used to approximate umbrellas... like in modelling)



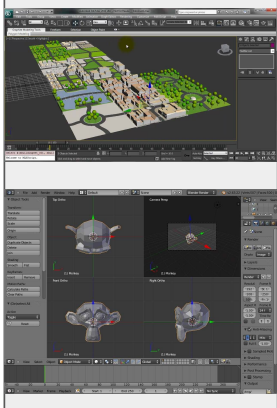
# Many other parts of scene setup!

- Volumetric fog
- Particle generators
- Decals
- Physics, destructibility, fluids, etc.

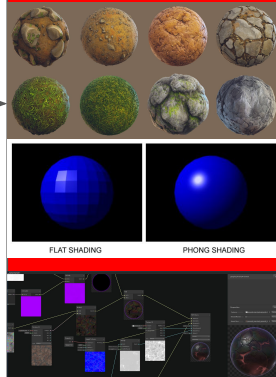


# Scene/Model Parameter Setup

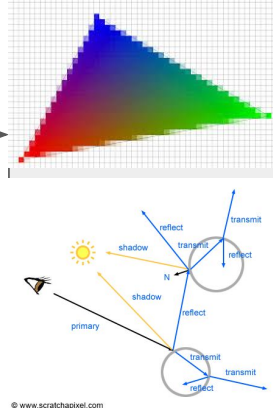
Scene Setup  
(Geometry,  
transforms, etc.)



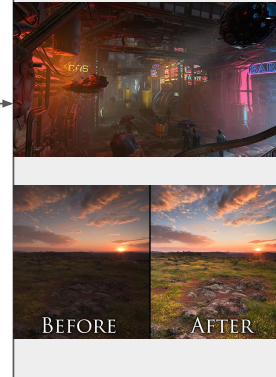
Scene/Model  
Parameter Setup  
(shaders,  
materials)



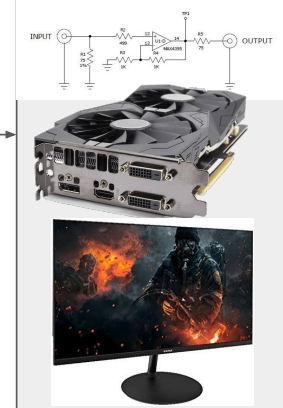
Rendering  
(Rasterization or  
Ray-tracing)



Post-processing  
(anti-aliasing,  
blur, color  
grading, etc.)

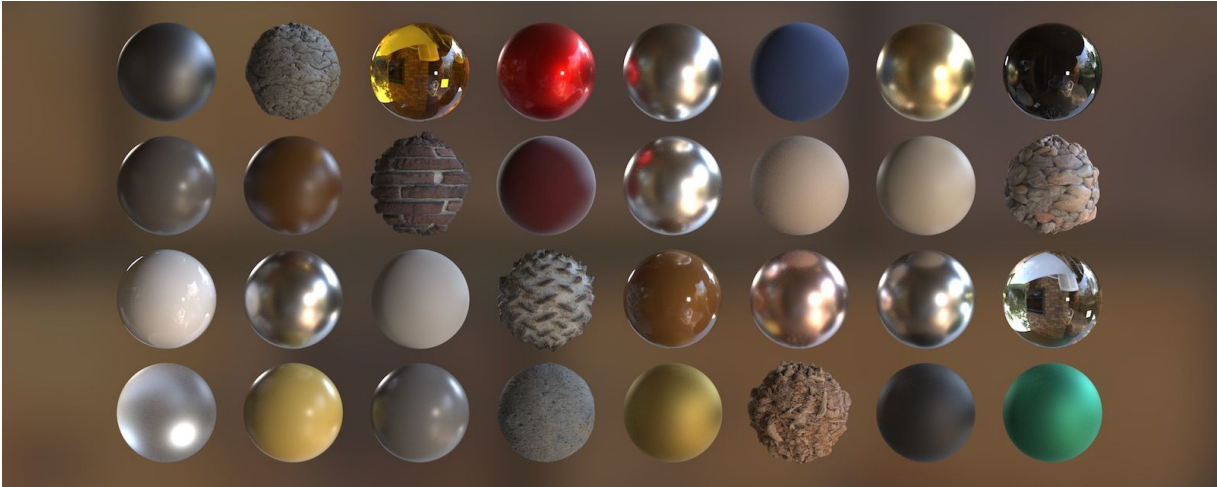


Output to  
Display (buffered  
output)



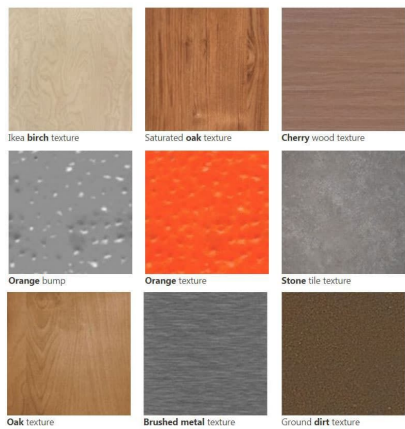
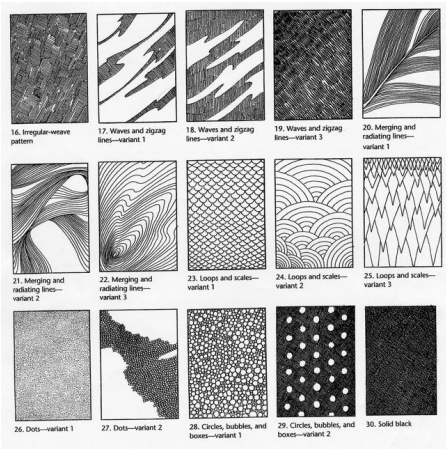
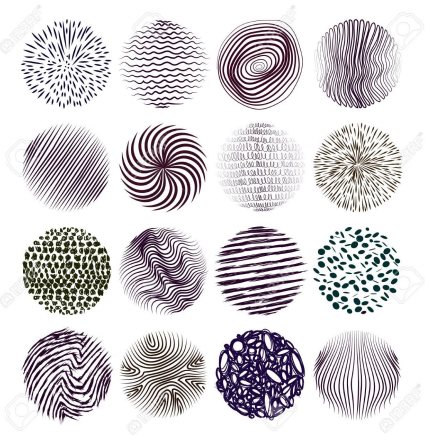
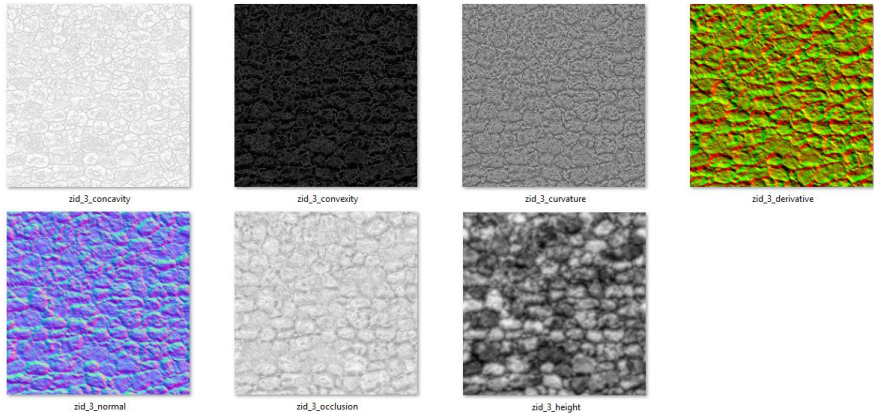
# Giving Details to the 3D Scene

- Need to tell **renderer** (which outputs the image):
  - How to show the 3D model (colors, textures, etc.)
  - How model interacts with scene, esp. lighting (reflections, absorption, etc.)
- **Materials** encode the model's parameters (textures, colors, smoothness, etc.)
- **Shaders** are mini-programs that tell renderer what to do with that info



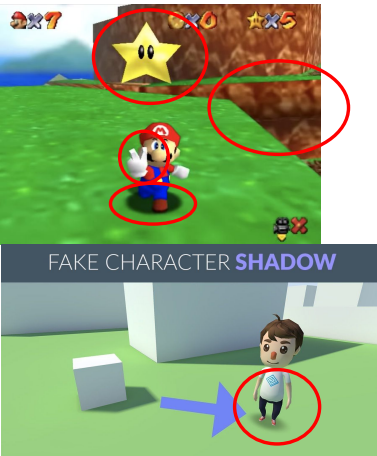
## Textures

- Images...as simple as that!
  - Usually .png, .jpg, .tga
  - Could specify that they're images used **before** rendering for **setup**
- Have different purposes



# Traditional 3D Graphics (90s)

- Computer hardware not strong enough to run in realtime
- Everything needed to be preprocessed & stored somehow
- Materials were basically just **textures** with various elements baked onto them with **texture maps** (at time drawn by artists!)
  - Back then, mostly shadows and bumps
  - Maps are still important optimizations
- **UV maps** used to apply the textures to 3D models

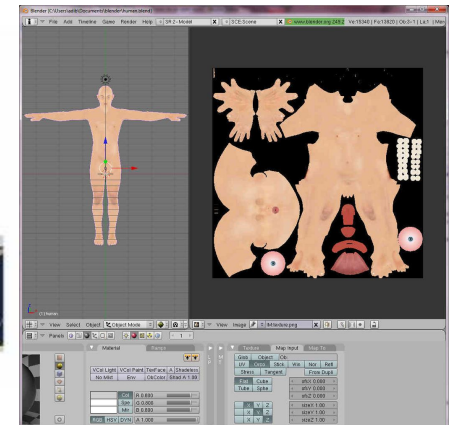


Great resource for understanding different maps:

<https://help.poliigon.com/en/articles/1712652-what-are-the-different-texture-maps-for>

## UV mapping

- Texture is 2D, model is 3D....how do we put texture on model?
- **UV mapping** is like wrapping a piece of paper (with image) around the model
- Often do it through the inverse example: **UV unwrapping**
  - flattening the model and overlaying the texture. Like origami!



We have a class at UNC on this called "Visual Solid Shape!"

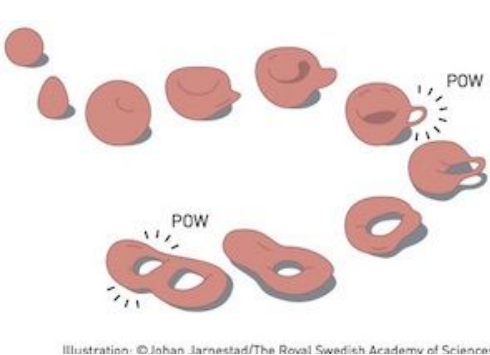
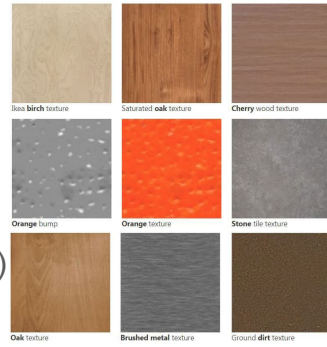
### • Optimization: UV/Texture atlas:

- mapping of many distinct texture/UVs of separate models/parts onto single texture/UV map
- In many cases, it's used to merge all textures in scene as one
- UV atlas is generally extremely high resolution

# Maps, maps, and more maps!

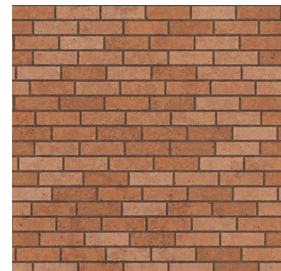
“Map” is used in so many contexts in game dev

- Can refer to images/**texture maps** with specific roles (diffuse, opacity, normal, etc)
  - Can fake effects as in reflection/specular & HDR maps
  - Can assist physically-based rendering (e.g. which part of the model is reflective?)
- Can refer to **mathematical** mapping
  - Topology
  - Mapping between coordinate systems (local & global, UV & model space, etc.)
- Can refer to **game maps**
  - Often small levels, like multiplayer maps
- Can be a **literal map** in your game!






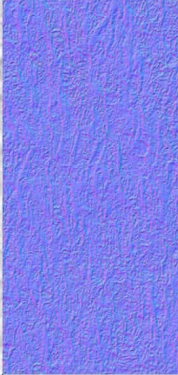


## Diffuse Map

- The surface details of the model without effect of light
  - Color
  - Texture
  - Patterns
  - Flaws, randomized features, etc.
- Anything besides solid colors start with a **texture**
  - Can be used as is, or transformed through **Material Functions**
  - Even solid colors usually treated like textures in game engines...4D RGBA Vector repeated per-pixel
- Often synonymous with **albedo** or **base color** (in Unity) but *technically* not the same in theory



# Some types of texture maps

BASIC GUIDE FOR TEXTURE MAP TYPES BY YANKO STEFANOV

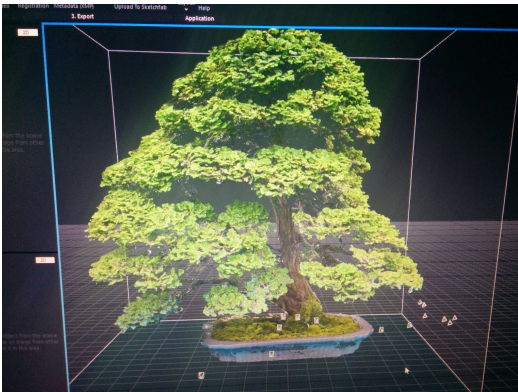
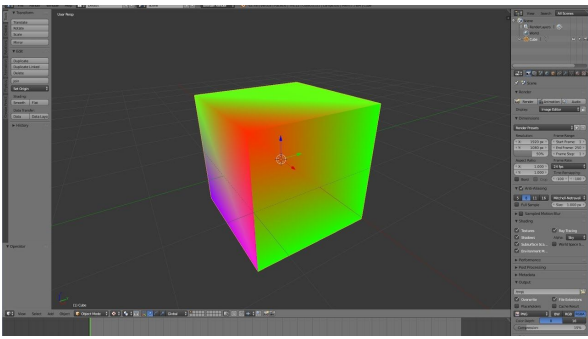
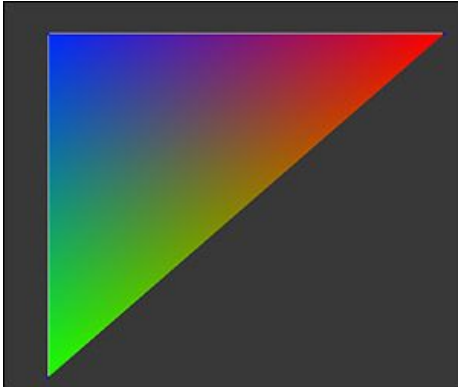
					
<b>DIFFUSE</b>	<b>SPECULAR</b>	<b>DISPLACEMENT</b>	<b>AMBIENT OCCLUSION</b>	<b>NORMAL</b>	
A diffuse map contains the color information of the texture. It defines what original color the texture will have at a certain position.	A specular map defines how strong the textured surface will 'shine' at a certain position. Most render engines use this information to define the appearance of specular highlights	Displacement maps are similar to bump but store height information and generates 'real' depth by adding geometry when rendered	Ambient occlusion is a method to approximate how bright light should be shining on any specific part of a surface. Place it on top of the Diffuse in "Multiplay mode" in Photoshop	The red, green, and blue channels of the image are used to control the direction of each pixel's normal. Normal map simulates the direction the normal is facing as well as the height, giving a more realistic effect.	Original model by: Zeljko Mihajlovic - 3D Artist

Great resource for understanding different maps:

<https://help.poliigon.com/en/articles/1712652-what-are-the-different-texture-maps-for>

## Another Cheap Method: **Vertex Colors**

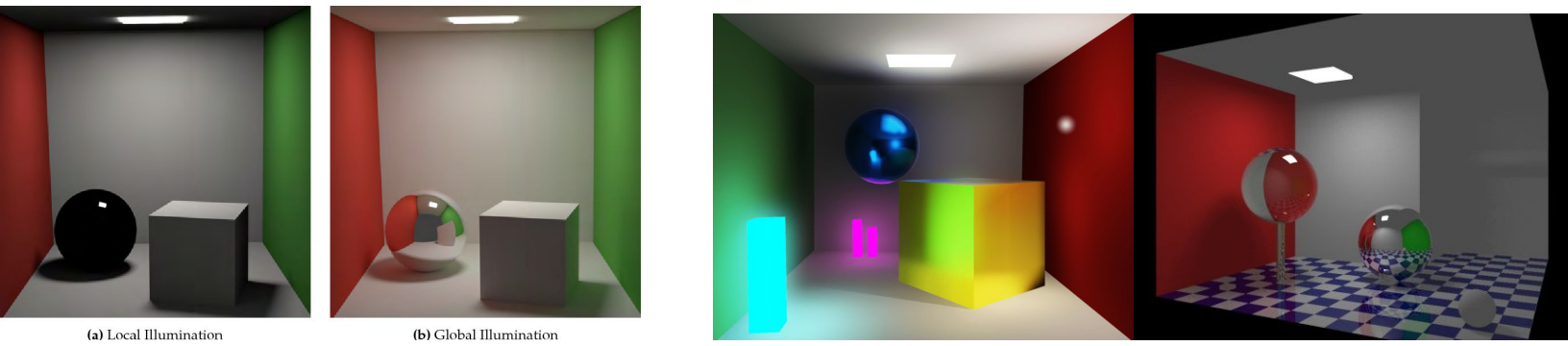
- Give each vertex of the triangle a color and linearly interpolate (**lerp**) along the polygon (if there is one)
- Very cheap and simple, but **major limitations**
  - What if the model has few triangles (low-poly)?
  - What about sharp changes in topology? Corners of a cube?
  - **Vertex colors** used for dense vertex-based models, e.g. 3D point clouds
  - **Textures** used for polygon-based models



Fun resource for more info: <http://www.alkemi-games.com/a-game-of-tricks-ii-vertex-color/>

# Nowadays...Physically-Based Rendering!

- **Light rays are predictable** as are most things in traditional physics
- We use **global illumination** (GI) to model lighting of a scene
- We use **physically-based rendering** (PBR) to model how meshes & their materials interact with GI and approximate the light paths
  - **Materials** include this description of interactions (smoothness, textures, etc.)
  - **Shaders** include info about getting everything to render and display (like little C programs...e.g. What does it **mean** for an object to have 0.75 smoothness?)
- Thus, **functions** can describe the light with parameters changed dynamically
  - e.g. player position/rotation, moving lights, varying brightness, deformed mesh

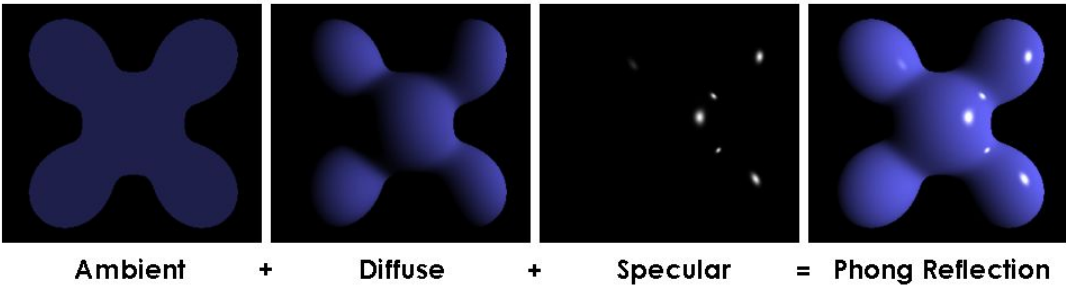


## Pre-PBR: Phong reflection model

- Start with texture/color & mesh and apply reflection model on top of it
- Reflection model: a function of **constants**
  - **Diffuse/Matte**: How much of the light's color survives
    - e.g. if light is blue & diffuse is high, a lot of the blue survives and makes model more blue
  - **Specular**: How much should light reflect and make the surface glossy
    - Maximum specular means you can only see reflection of scene like HDR map
  - **Ambient**: Base amount of light applied evenly throughout scene



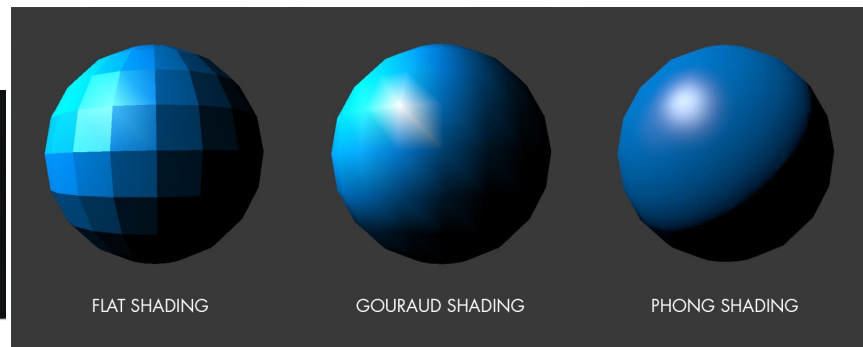
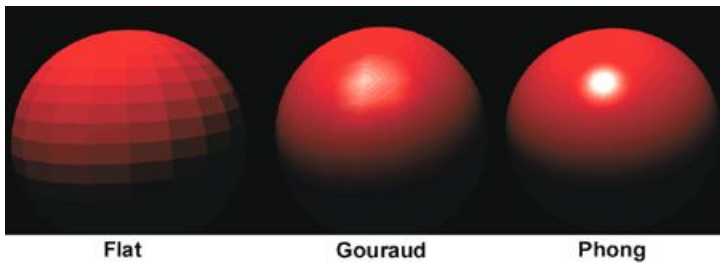
- Improved with Blinn-Phong model
- Still in use today and is **de facto baseline** for 3D shaders
- Not quite PBR since parameters are constants...PBR describes them as **functions**
- Thus, rough estimation
- Only option in Unity until HDRP



Ambient + Diffuse + Specular = Phong Reflection

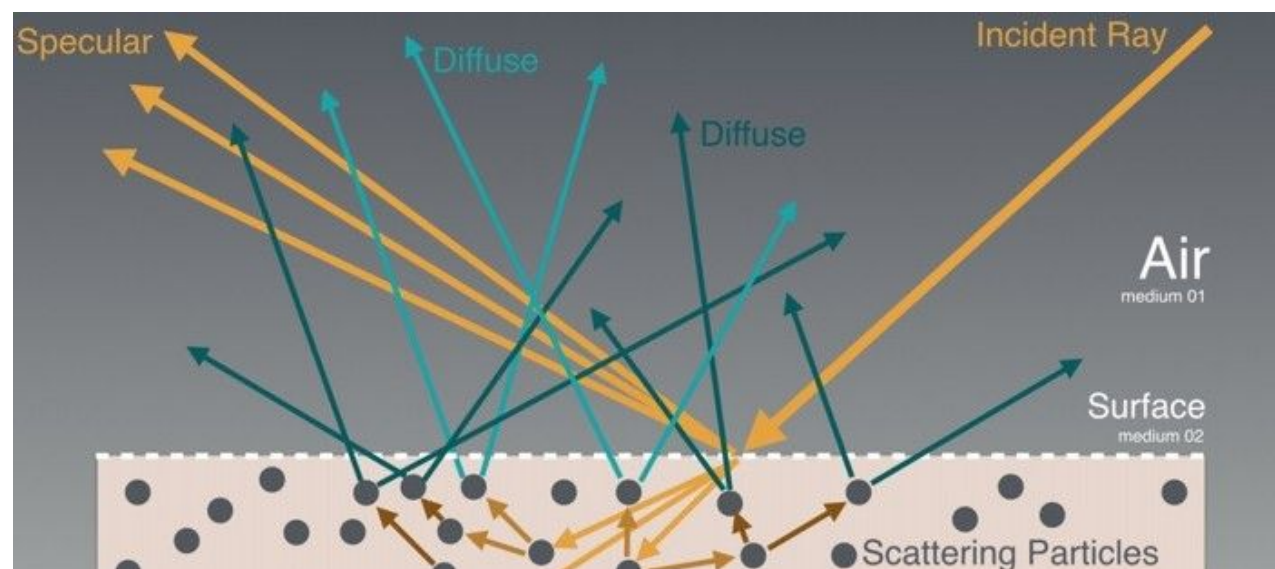
# Quick History of 3D Shading

- Methods of **interpolating** model edges when rendering image
  - Can make model **appear smooth without geometrically smoothing** it (e.g. subdividing)
  - Visual trickery for a great optimization!
- Such methods often called “**smooth shading**”...compare to “**flat shading**” below
- One of the first smooth shading methods: **Gouraud shading** (1971)
  - Lerps between vertices...similar to vertex colors in concept
  - Massive contribution in computer graphics...allowed rendered models to have curves with few verts!
- Another major contribution from Phong: **Phong shading** (1973)
  - Allows for interpolation **WITH** specularity!
  - Still a common method!



## Basics of PBR

- **Incident ray**=light ray
- **Diffuse** reflections=rays that get scattered (detail of model that you see)
- **Specular** reflections= rays that reflect the environment (ooooo shiny!)
- Sometimes we model **medium** (e.g. passing through water or glass)



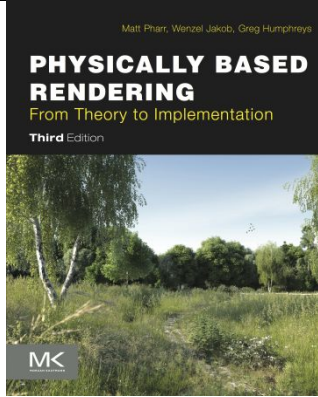
From “The Comprehensive PBR Guide” by Algorithmic

# How to actually implement PBR?

- At first, it was mostly just mixing a bunch of lighting models together, such as:



Found on <https://theovermare.com/blog/2015/02/the-journey-of-the-light-physically-based-shading/>



- In game engines, it's much more complex but unnecessary to know the details unless you work that low-level

## Shader Languages

### GLSL

- OpenGL Shader Language
- Similar to C
- Usually only used if interfacing with OpenGL directly

```
void main(void) {  
    mat4 modelView = MVMatrix * XformMatrix;  
    vec4 ecPosition = modelView * MCVertex;  
    gl_Position = PRMatrix * ecPosition;  
    diffuseTextureCoord = TexCoord0;  
  
    if (EnableLighting)  
    {  
        ecPosition3 = (vec3(ecPosition)) / ecPosition.w;  
        ecNormal = vec3(modelView * vec4(MCNormal, 0.0));  
        if (EnableNormalize)  
            ecNormal = normalize(ecNormal);  
    }  
}
```

### HLSL

- High-level shader language
- What Unity, Unreal, and most other high-level APIs use & expose to dev
  - Unity HDRP & UE4 abstract them
- Still pretty similar to C....more like C++

### Structure of HLSL code

- HLSL code has four parts:
- Variables that get values from application
  - Input and Output structure (optional)
  - Functions
  - Techniques and passes.
- One HLSL file could have more **techniques**. One **technique** can have multiple **passes**. Lets see the following example

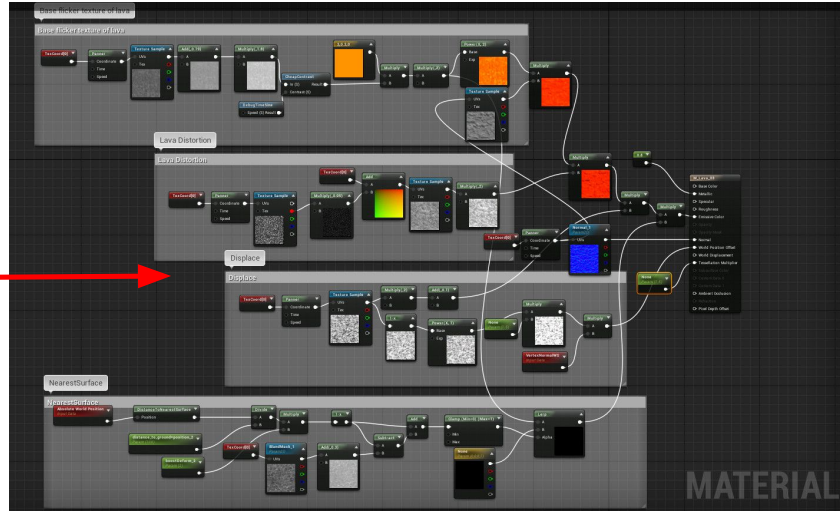
```
struct VS_OUPUT  
{  
    float4 pos          : POSITION;  
    float4 color        : COLOR0;  
};  
  
float4x4 WorldViewProj : WORLDVIEWPROJECTION;
```

Rarely need to touch either one nowadays unless making shaders from scratch

# Emergence of Shader Graphs

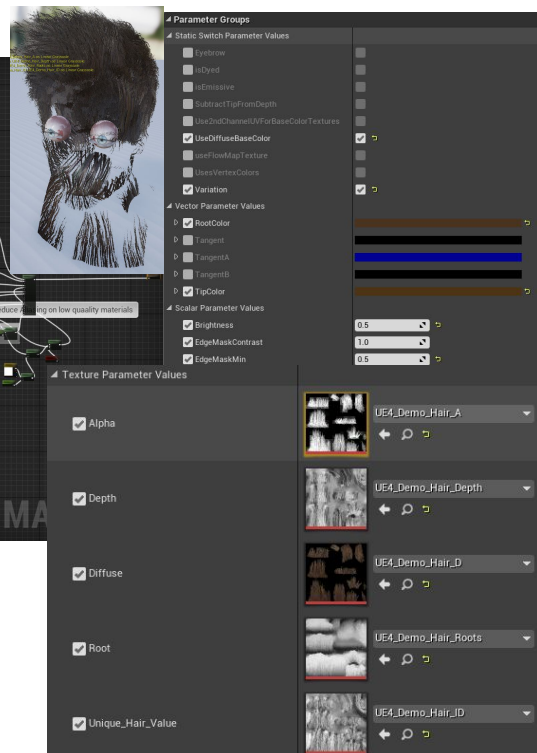
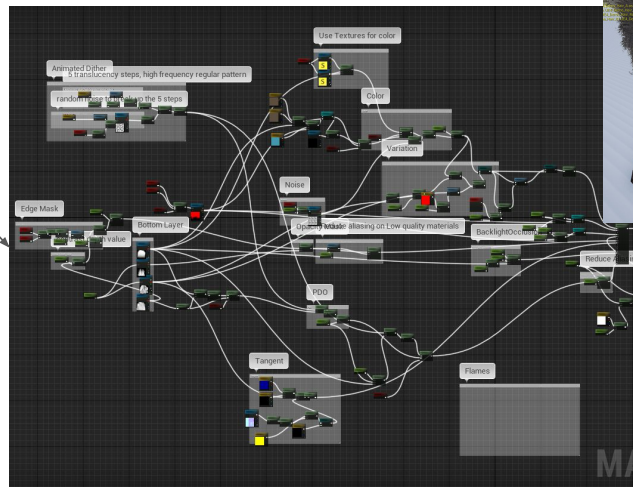
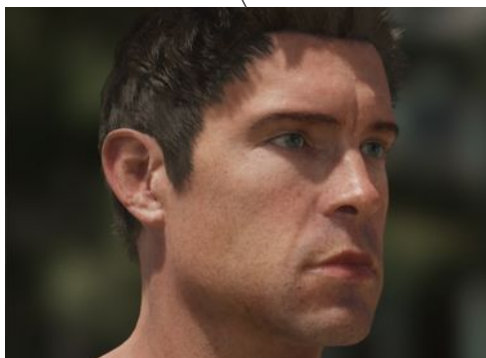
- Shading more accessible to high-level devs.... Like game devs!
- Results are immediately apparent & can be displayed visually
  - Why wouldn't we want to display graphics-related concepts graphically if possible?

```
source.tmp.hlsl Graphics Experiment.vsglog* Graphics Experiment.vsglog*
float4 pos : SV_POSITION;
float2 pack0 : TEXCOORD0;
float2 lmap : TEXCOORD1;
LIGHTING_COORDS(2,3)
};
#endif
#ifndef LIGHTMAP_OFF
float4 unity_LightmapST;
#endif
float4_MainTex_ST;
v2f_surf vert_surf (appdata_full v) {
v2f_surf o;
o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
o.pack0.xy = TRANSFORM_TEX(v.texcoord, _MainTex);
#ifdef LIGHTMAP_OFF
o.lmap.xy = v.texcoord1.xy * unity_LightmapST.xy + unity_LightmapST.zw;
#endif
float3 worldN = mul((float3x3)_ObjectToWorld, SCALED_NORMAL);
#ifdef LIGHTMAP_OFF
o.normal = worldN;
#endif
#ifdef LIGHTMAP_OFF
float3 shlight = ShadeSH9 (float4(worldN,1.0));
o.vlight = shlight;
#endif
#ifdef VERTEXLIGHT_ON
```



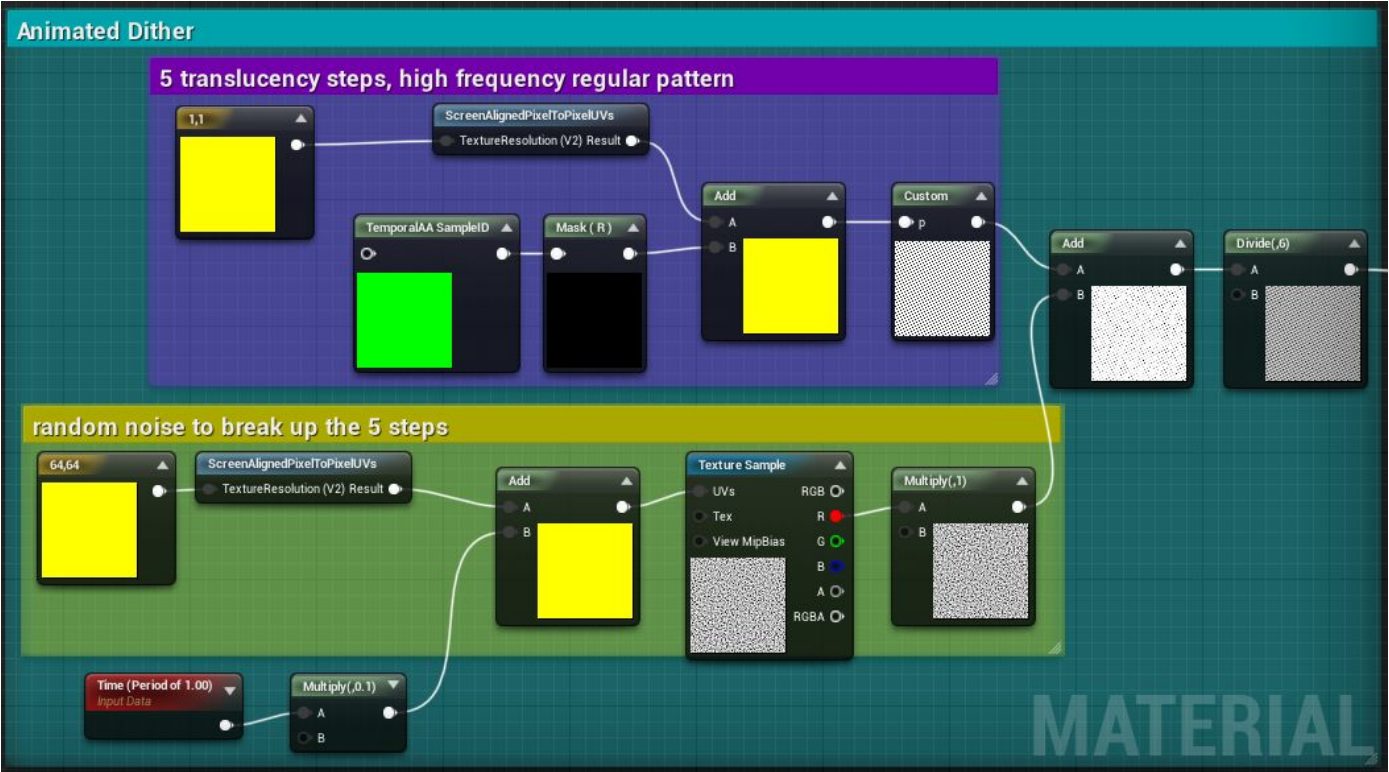
# PBR & Material Functions (Composite of Shaders)

- PBR enables all materials to be parameterized functions with realtime light response
  - Powerful for randomization, dynamic materials, etc.



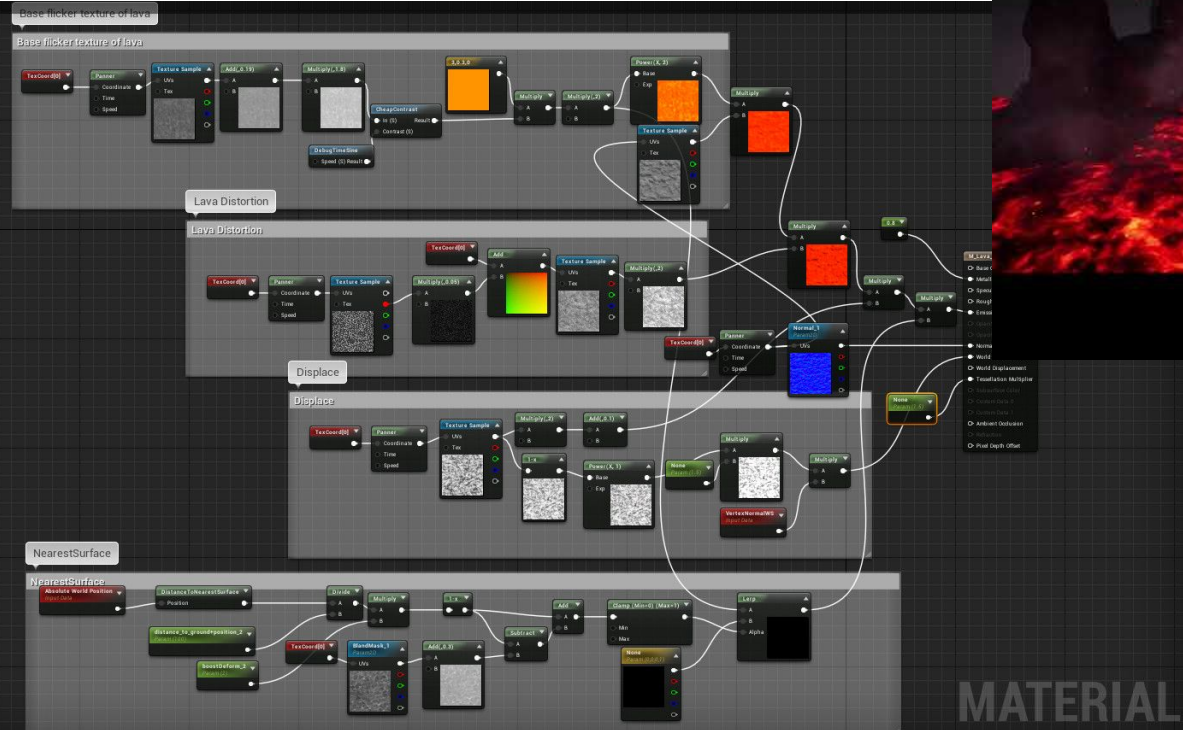
Randomized hair material in UE4 from Digital Human demo

# PBR & Material Functions



Small portion of that previous material randomizing small, periodic motions in the hair

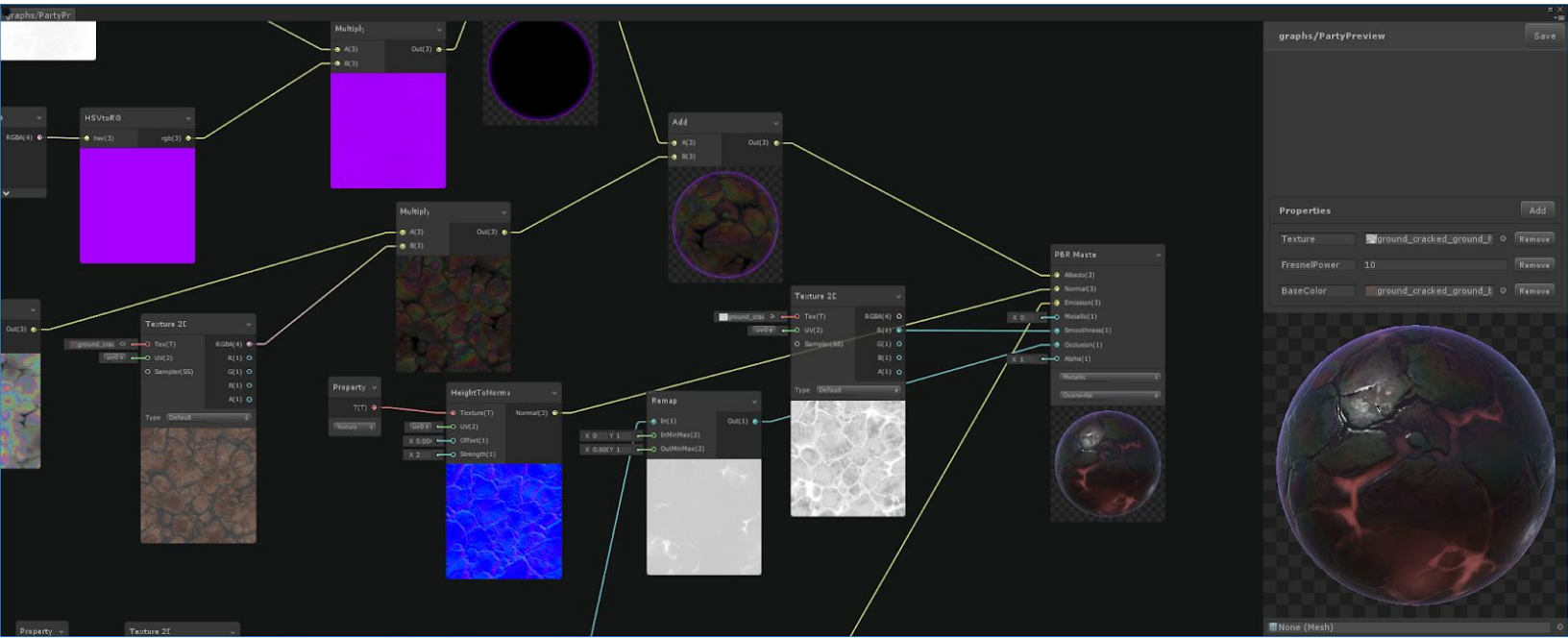
## Another example



- Lava Effect by Tanya Jeglova on <https://www.artstation.com/artwork/mqAk0Z>
- Nice tutorials at <https://www.youtube.com/watch?v=H13BbNvKYjA> and <https://www.youtube.com/watch?v=b1vjz3A3anQ>

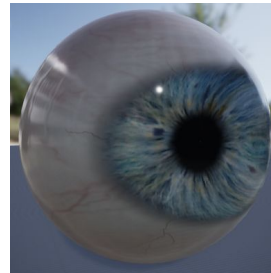
# Unity 2018 Shader Graph

- They added a graph similar to UE4's
  - Not fully featured but they're getting there...
  - At least they're moving on from Phong!

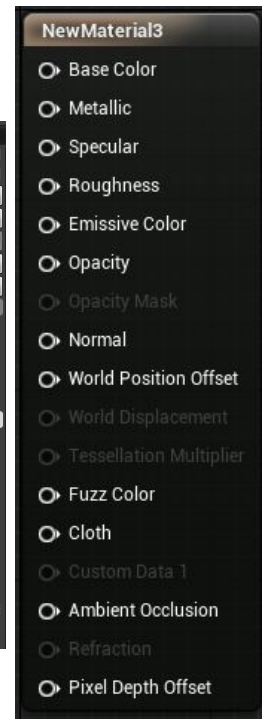
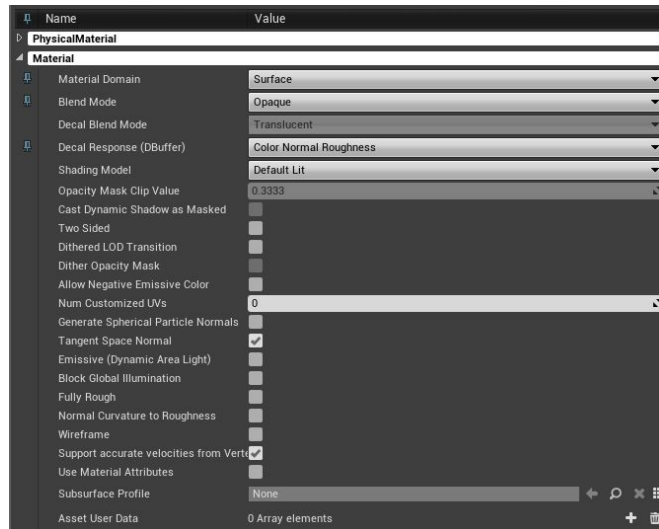
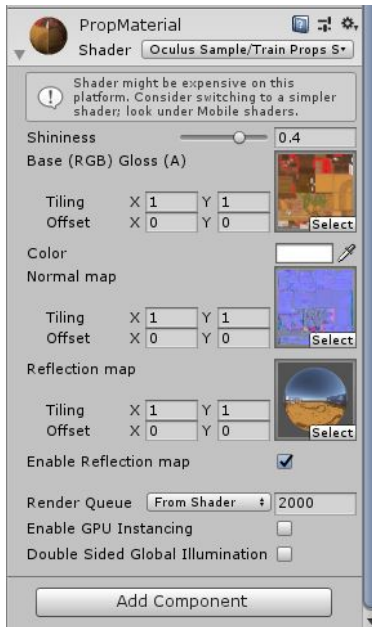


## Material Properties

Unity  
(non-HDRP)



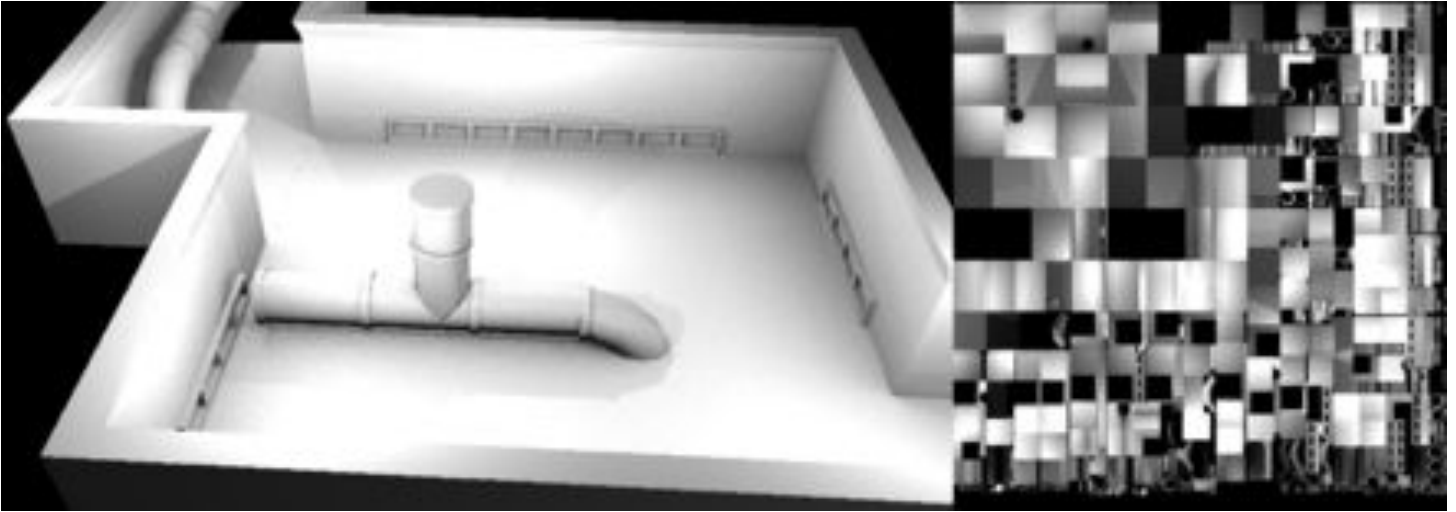
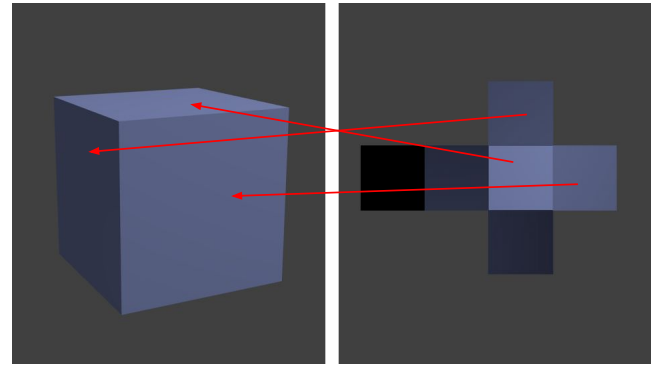
UE4



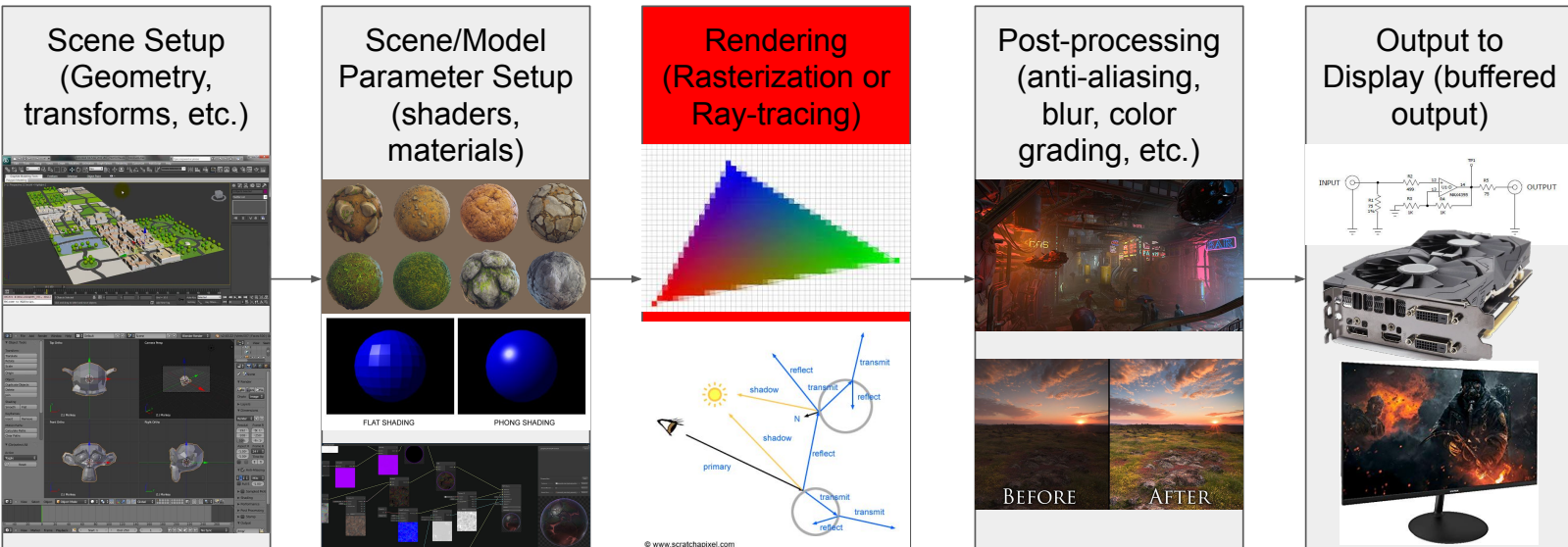
(small subsection of UE4 material properties)

# Light Parameters & Lightmaps

- Radiance/luminosity/intensity
  - lux, candelas, lumen
- Dynamic vs static/baked lighting

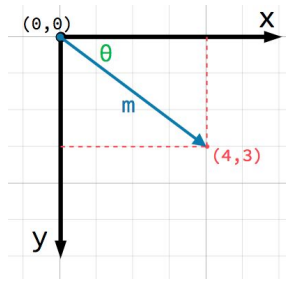


# Rendering: Creating the Image

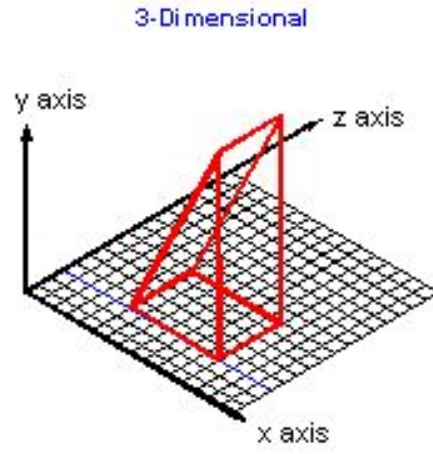
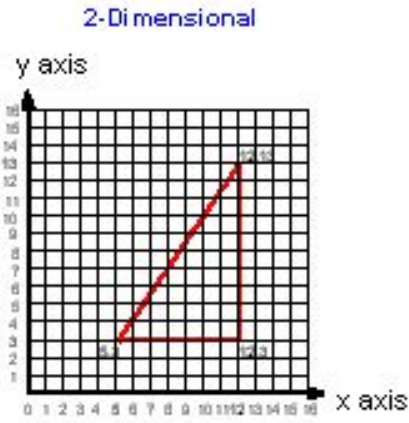
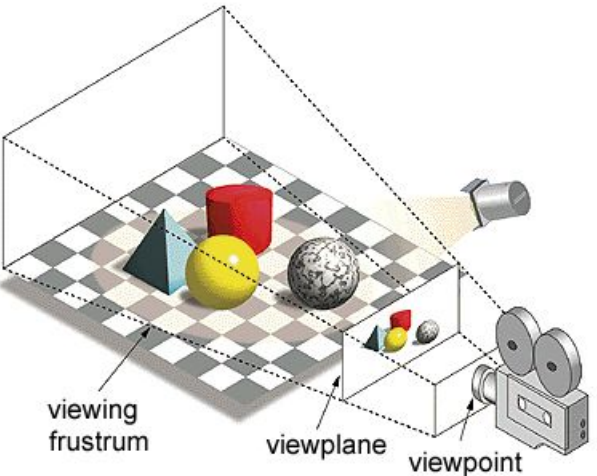


# Cameras

- Structure representing viewpoint... Virtual implementation of physical camera
- **Camera plane:** reference plane used to create image
  - like world origin of the 2D rendered image!
- **Camera frustum:** camera's range of vision

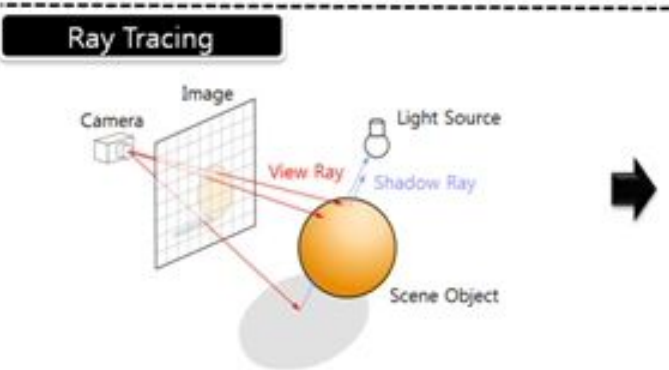
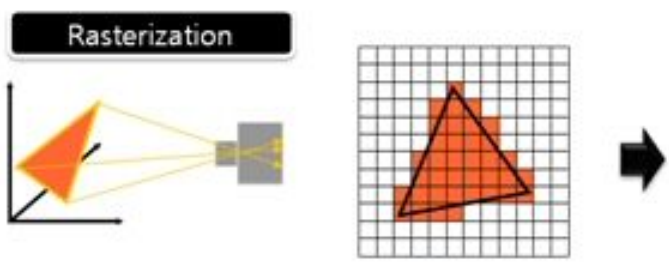


From Computer Desktop Encyclopedia  
 Reprinted with permission.  
 © 1998 Intergraph Computer Systems



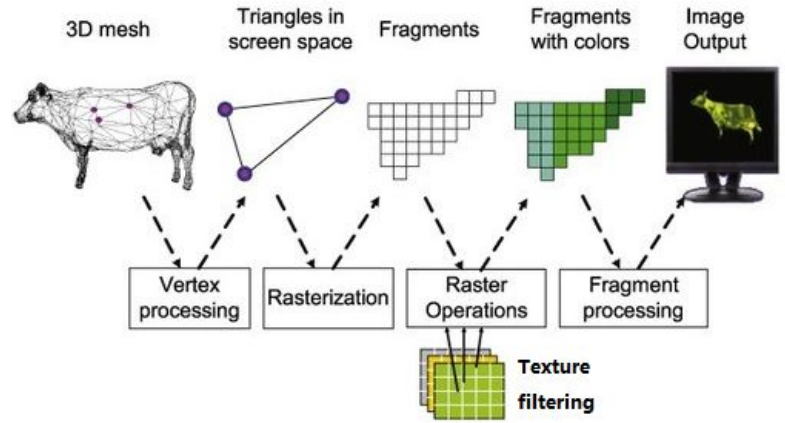
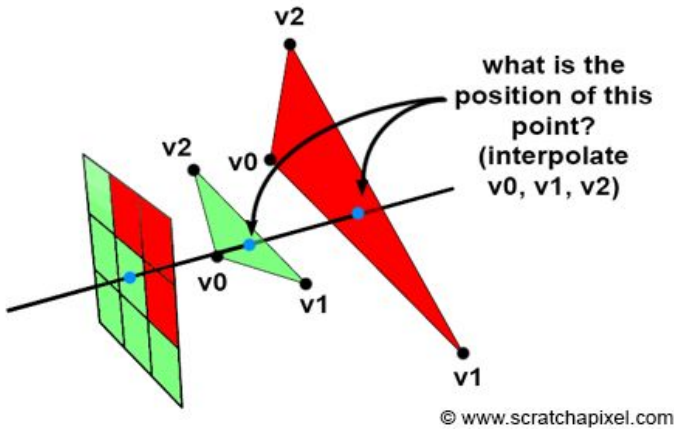
# Two Major Rendering Methods

- Rasterizing
- Ray-tracing
- **Main difference:** how you learn the source of a pixel



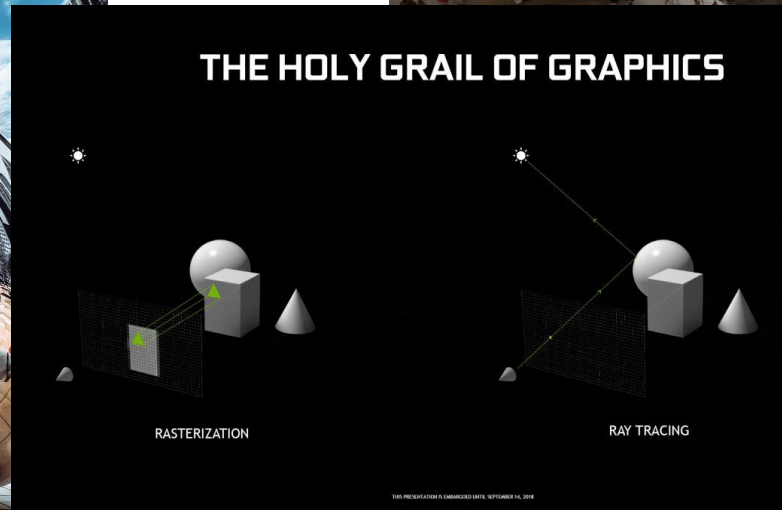
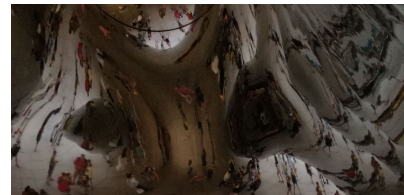
# Rasterizing

- Uses **z-buffer** to determine layer that each slice of 3D scene is on
  - Like dividing 3D scene into **cross-sections** parallel to camera plane
- Fast and default rendering method, essentially just projects pixel to camera plane



## Where Rasterization Fails

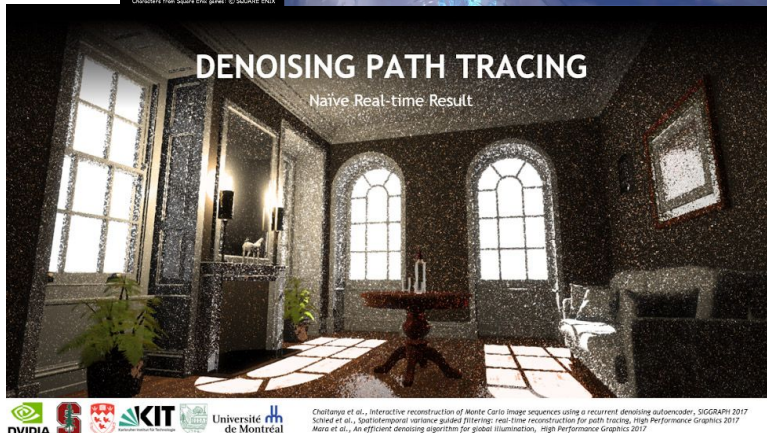
- Can Cloud Gate, Chicago be rendered with a rasterizer? What would it look like?
  - Reflected object is seen from a different angle from the forward vector of camera to the mirror...it comes from a vector from mirror to reflected object.
  - Rasterizer mostly just cares about direct rays of light...pixel doesn't "travel"
  - Only **rays** can accurately represent this



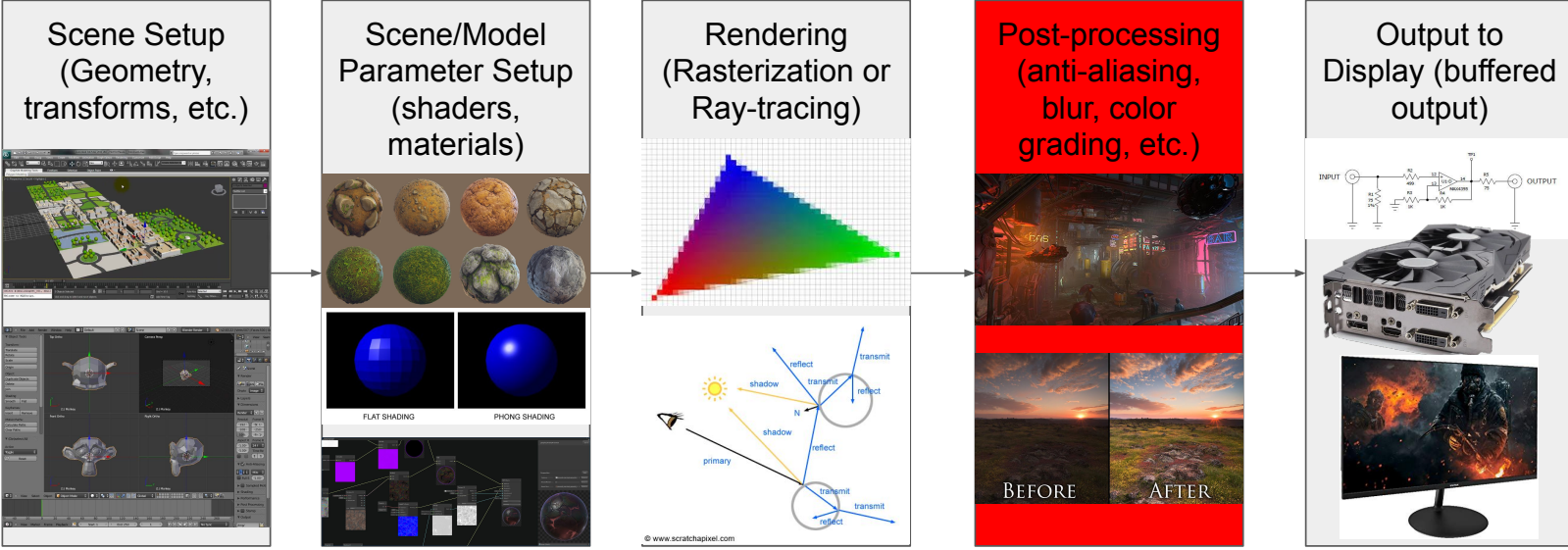
# Ray-Tracing (simpler Path-Tracing)

(probably not ray-traced... easier methods for planes! But imagine this for every surface)

- Learns pixel by shooting **rays** from lights & cam
  - Gives a better impression of the 3D scene
- Much slower than rasterizing...rays are harder to compute than pixels. Z-buffer is like precomputing
- **Denoising** is making ray-tracing more feasible
  - Denoising basically fills in the blanks, requiring fewer rays

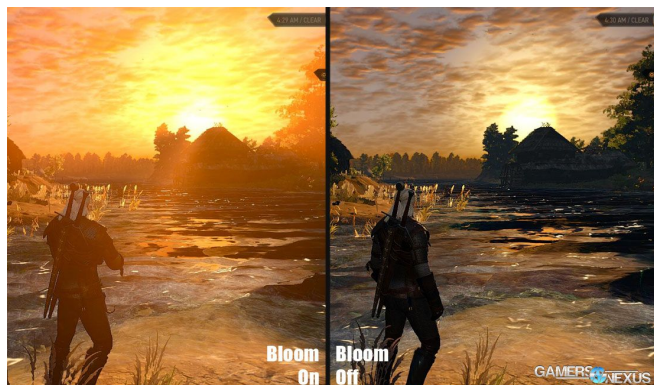


# Post-Processing



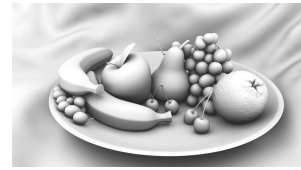
# Purpose of Post-Processing

- 3D environments are complex & require specialized algorithms
- 2D image processing is really fast on modern GPUs
- So add some effects AFTER image is rendered from 3D scene
  - (which # pass depends on whether it's **deferred** or **forward** rendering)
- Lots of beautification can be done in 2D with simple image processing
- Often called **post-processing pass** or **post-processing layer**
  - Each pass is a different set of effects applied

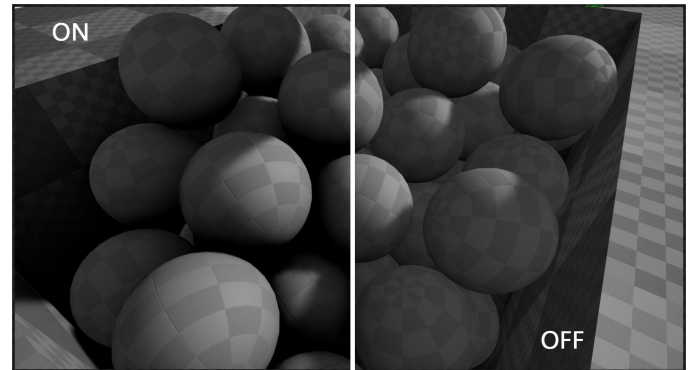
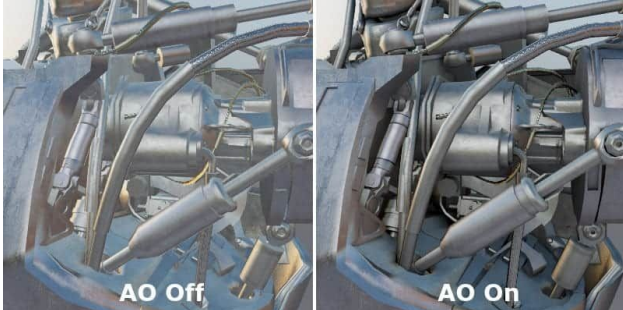


## Some Common Post-Processing Options

# Ambient Occlusion

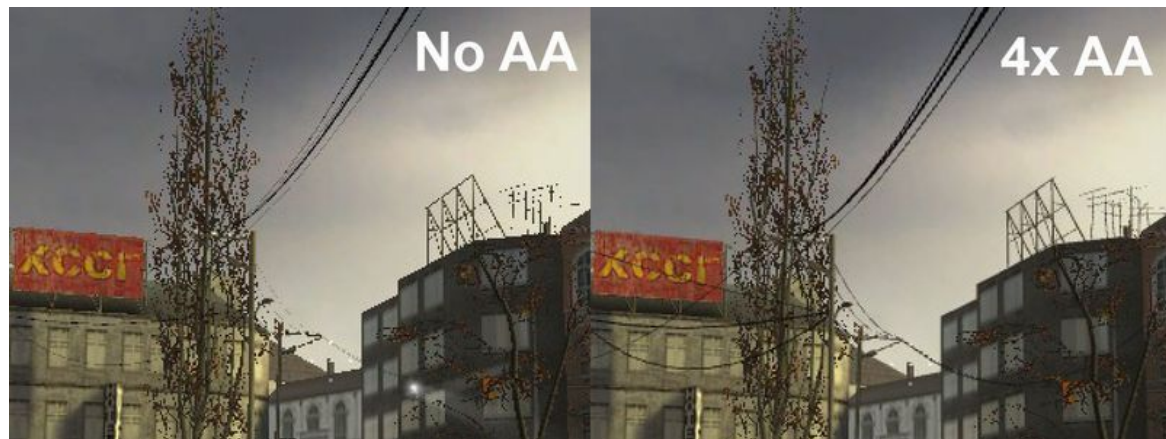
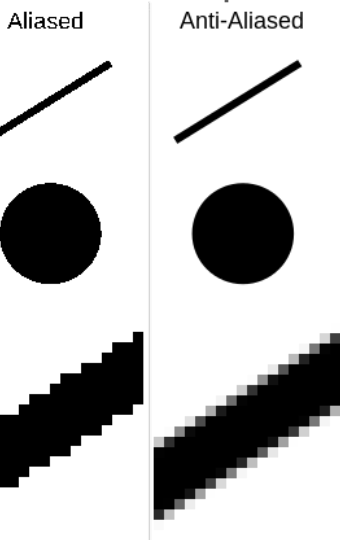


- Draw shadows where sudden change in topology, regardless of light
  - Estimating where shadows will probably be given corners & blocking objects
  - Approximates real ambient light instead of adding luminosity to everything
- Gives exaggerated sense of depth



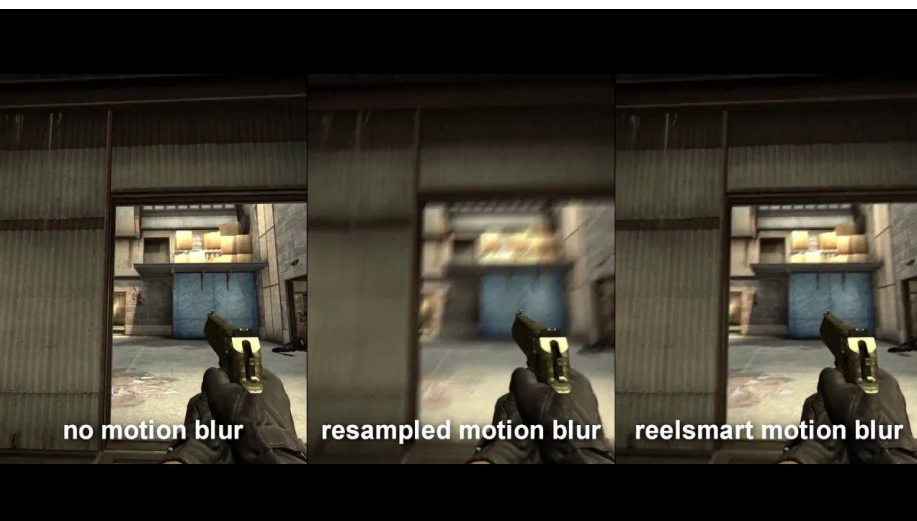
# Anti-aliasing

- Aliasing: “jaggies” from limited # pixels
- Anti-aliasing: smoothing jaggies, usually by interpolating or filtering
- Can be per-frame or temporal



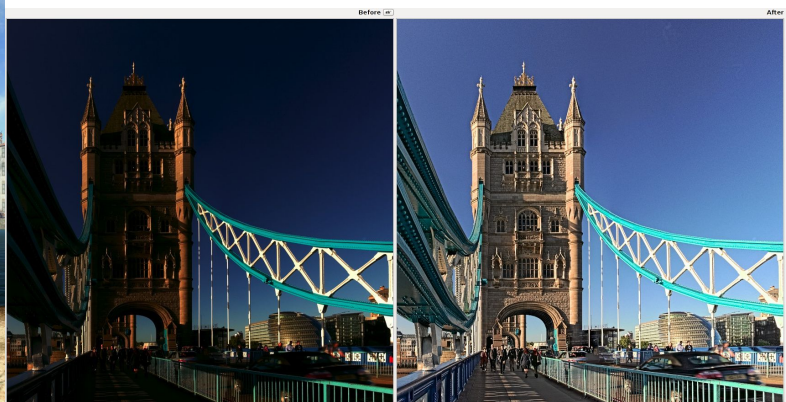
# Motion Blur

- Blurs objects moving faster than framerate can keep up with
  - Can stylize action sequences and things that are hard to make high-res (like grass)
  - In games, usually used to obscure low framerate
  - We try to avoid it in VR b/c it causes sickness



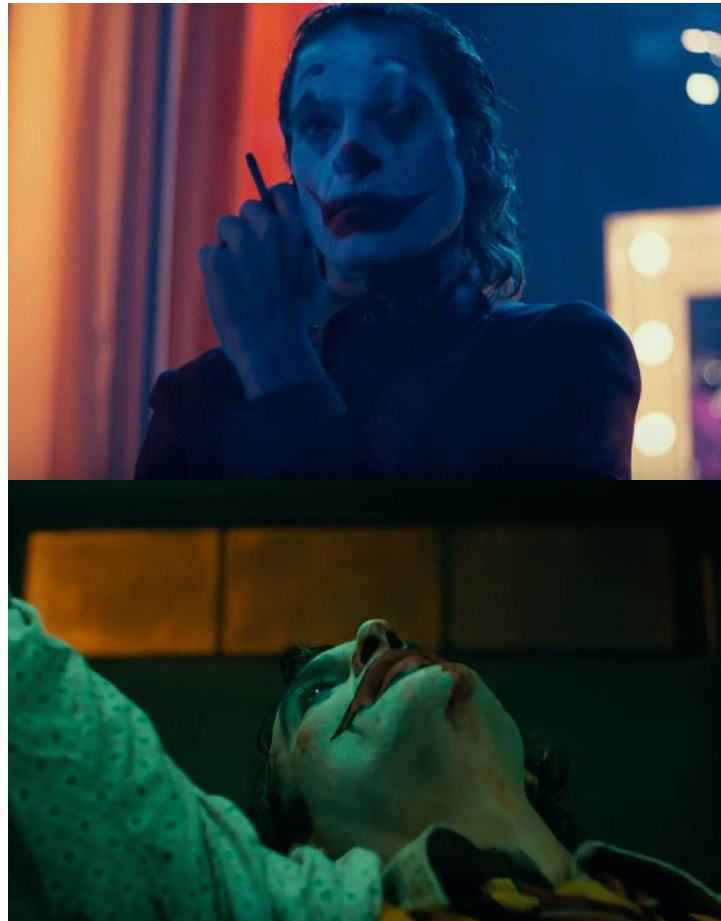
# Tonemapping

- Maps current color range to another...often faking HDR
  - Sometimes (like in UE4) make colors more natural...
    - E.g. pure white is almost nonexistent in real world, so map it to a pale color

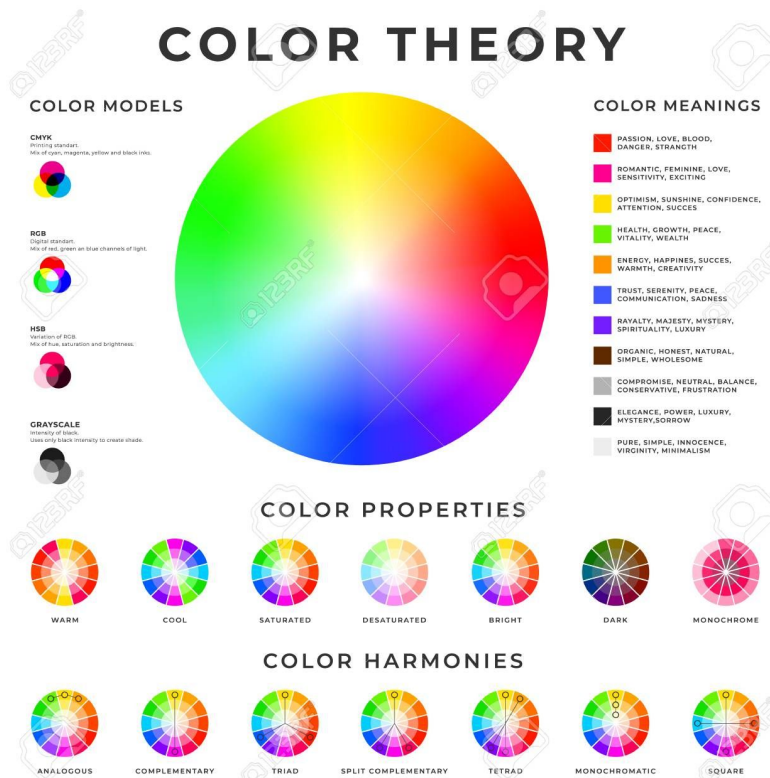


# Color-grading

- Changing color, gamma, brightness, etc. parameters to achieve stylistic effect

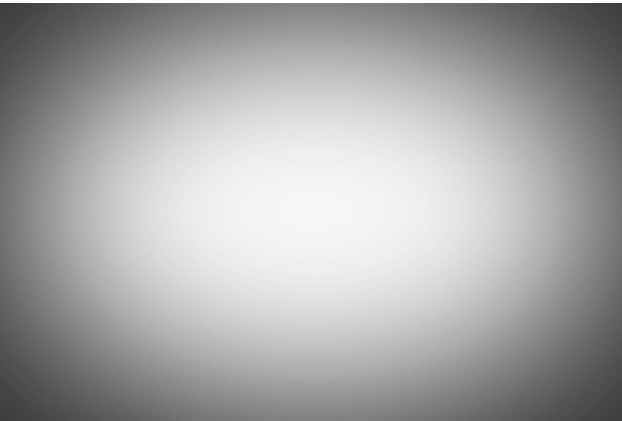
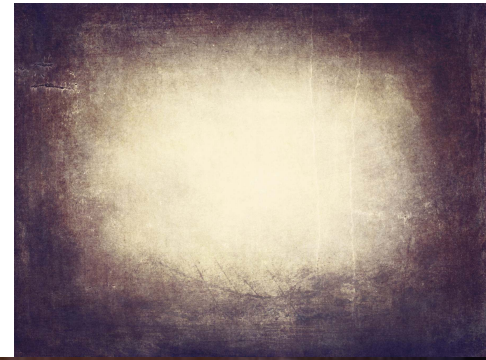


## Supplementary Material on Color & Tone



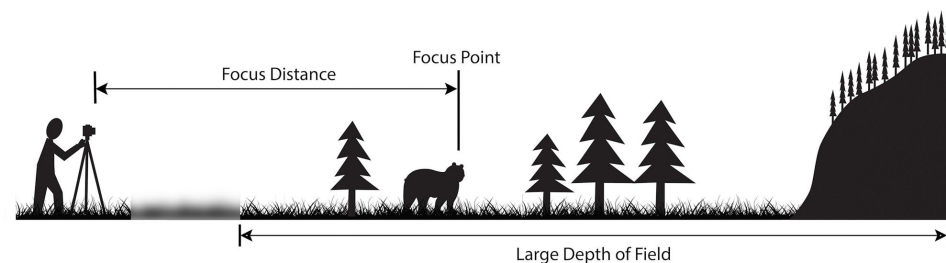
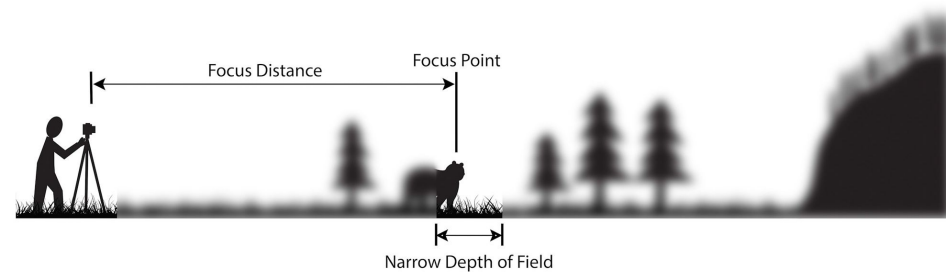
# Vignette

- This radial effect that looks like paper degradation or tunnel vision



# Depth of field

- Defocus things outside of focal range



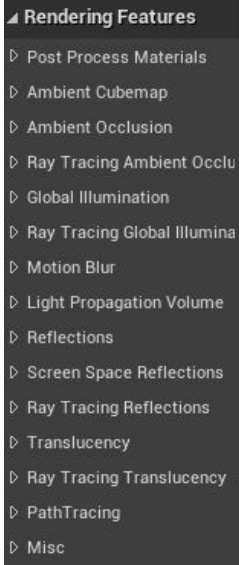
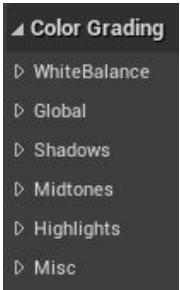
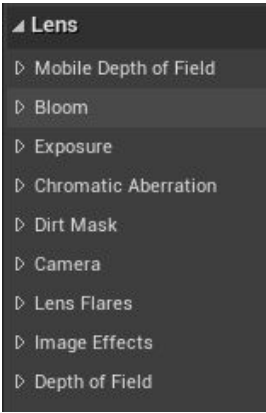
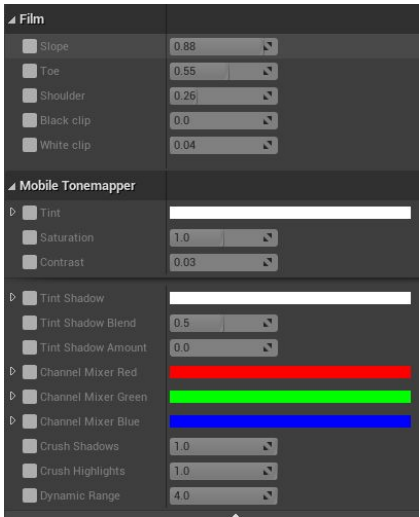
# Post-Processing in Game Engines



- Unreal 4 has always had a “**post-processing volume**” with a huge list of params. Can apply different post-processing to different areas of scene
  - Makes UE4 suitable for film CGI and architectural visualization (archviz)
- Unity 2018 added a “**post-processing stack**” with these volumes as well
  - Still has very few features...just common ones like anti-aliasing, bloom, etc

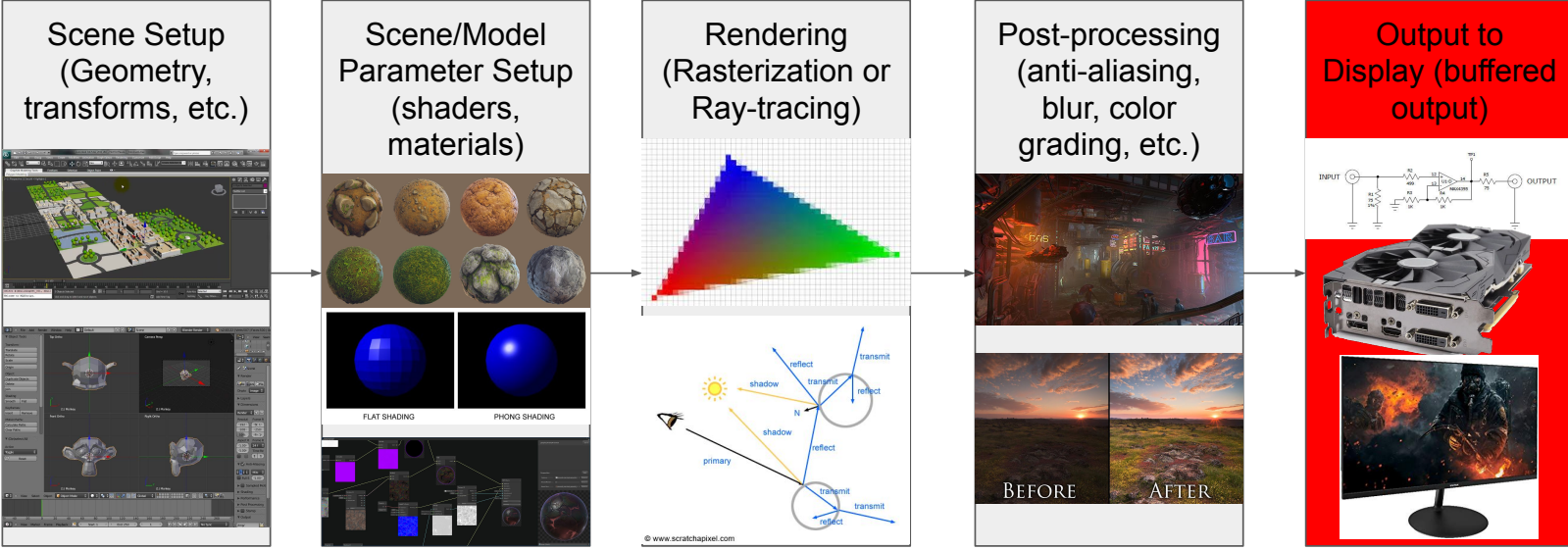


- Ambient Occlusion
  - Auto Exposure
  - Bloom
  - Chromatic Aberration
  - Color Grading
  - Depth of Field
  - Grain
  - Lens Distortion
  - Motion Blur
  - Screen-space reflections
  - Vignette
- Unity post-processing options as of 2019.2



Some of Unreal 4's options

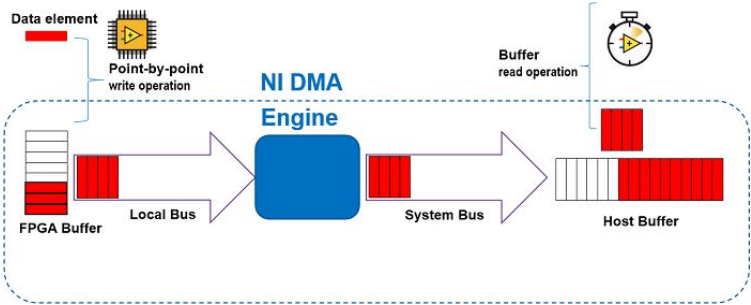
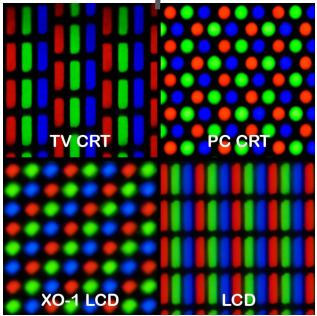
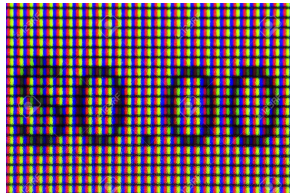
# Output: Showing the Image



# From Image to Screen

(above-average student bank account) ---->

- Some low-level API sends the image to the GPU, which handles output to device (fragments->pixels, etc.)
- Mentioned b/c older VR devices were treated like multi-monitor setups...nowadays we can tell which output is VR
  - OpenXR standardizes the HMD drivers



# Optimization & Complexity

# How do we work with limited hardware?

- Game devs already had to optimize for multiplatform
- Now we have all these VR devices (some mobile like Quest)
- What to do?

Legend of  
Zelda: Breath  
of the Wild



The Witcher 3



Uncharted 4



Battlefield 1



## Basic principle of complexity

The more complex the individual objects in a scene are, the fewer we can have!



# Importance of Complexity

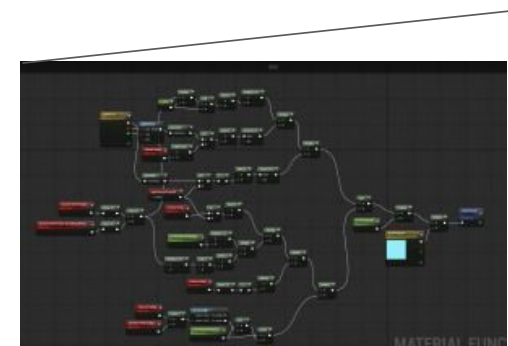
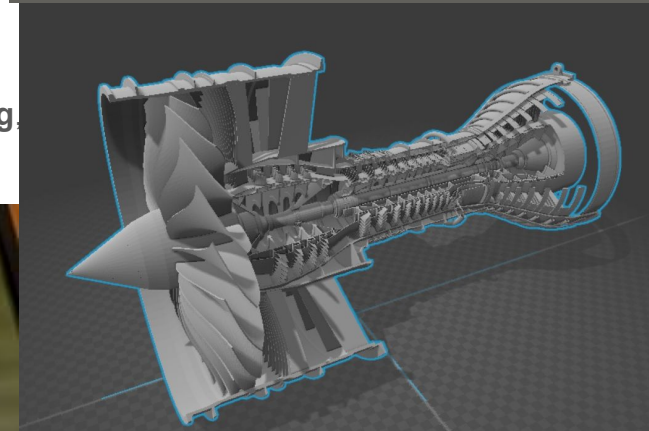
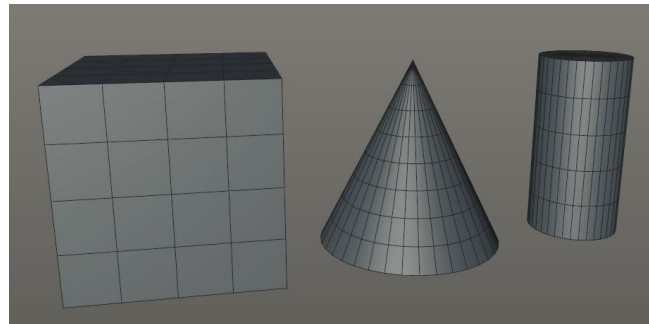
- Processing times
- Rendering load/times
- Memory usage (GPU and RAM)
- Affects number of objects in scene (scene complexity)



GPU [STATGROUP_GPU]	Average	Max
Counters		
[TOTAL]	13.76	14.30
Basepass	3.82	4.25
Postprocessing	2.52	2.89
Prepass	0.96	1.35
Reflection Environment	1.15	1.52
Capsule Shadows	0.85	1.24
HZB	0.40	0.43
Slate 3D	0.65	0.91
Lights	0.67	1.07
Translucency	0.56	0.97
[unaccounted]	0.45	0.84
Slate UI	0.34	0.75
Composition BeforeBasePass	0.23	0.28
Render Velocities	0.23	0.55
Particle Simulation	0.21	0.23
ShadowProjection	0.16	0.18
Distortion	0.16	0.56
Shadow Depths	0.14	0.23
Composition PreLighting	0.16	0.56
Custom Depth	0.09	0.24
Composition PostLighting	0.00	0.00

## What makes an object complex?

- Size relative to camera
- Vertex count
- Shape (affects shadows)
- Collision and contact complexity
- Resolution of maps (UV maps, lightmaps, etc)
- Intended materials
  - (eg. a human body part might use **subsurface scattering**, which is very computationally complex!)



# Complexity in Games



## System Requirements

### Windows

#### MINIMUM:

OS: Windows® 7/Vista/XP

Processor: Intel® Core™ 2 Duo E6600 or AMD Phenom™ X3 8750 processor or better

Memory: 2 GB RAM

Graphics: Video card must be 256 MB or more and should be a DirectX 9-compatible with support for Pixel Shader 3.0

DirectX: Version 9.0c

Storage: 8 GB available space



- OS: 64-bit Windows 7, Windows 8.1 and Windows 10
- Processor (AMD): AMD FX-6350
- Processor (Intel): Core i5 6600K
- Memory: 8GB RAM
- Graphics card (AMD): AMD Radeon HD 7850 2GB
- Graphics card (NVIDIA): nVidia GeForce GTX 660 2GB
- DirectX: 11.0.Compatible video card or equivalent
- Online Connection Requirements: 512 KBPS or faster Internet connection
- Hard-drive space: 10GB

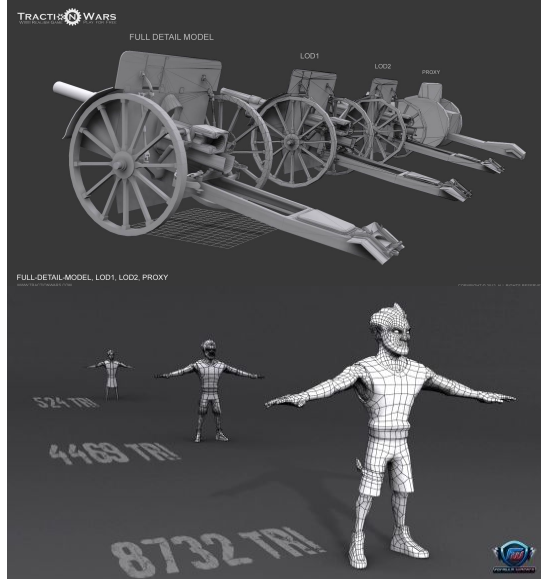
## How do we simplify complex objects?

- **Decimation** of vertices/recalculation of triangles
- **Maps**
  - Use when material functions unnecessary
  - Keep just high enough resolution to save RAM
- **Simplifying Shaders & Material Functions**
  - Avoid unnecessary computation
  - Share values (e.g. UV coordinates)
- **Level of Detail (LOD)**
- **Randomization** of certain details
- **Accuracy** parameters (shadows, textures, etc)

Save complexity for more important objects! (main characters, things that will be closer to the screen, etc.)

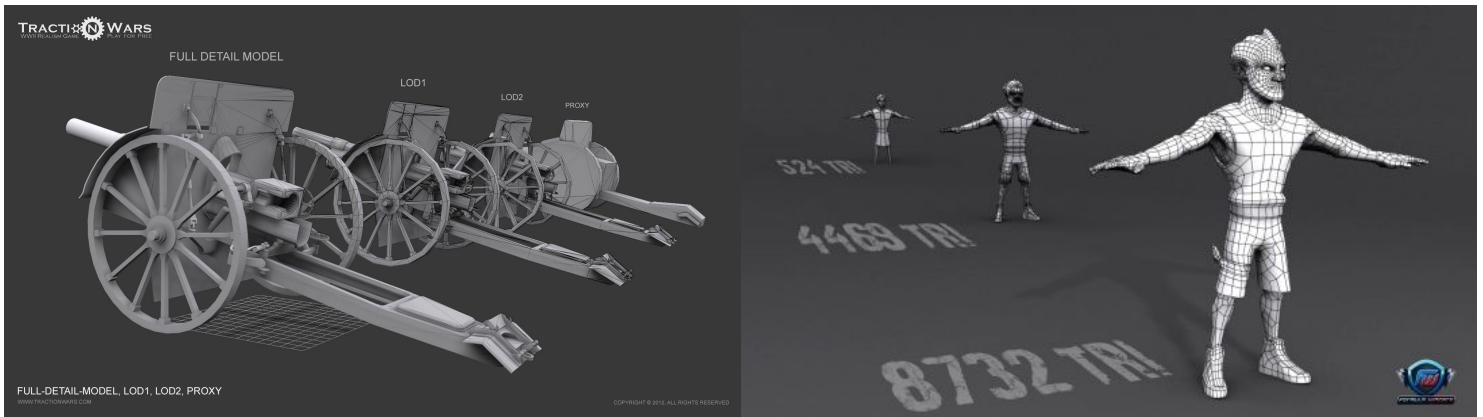
# Level of Detail

- Farther objects are, less detail they should have
- Great and common optimization
- Multi-platform almost impossible without it nowadays
- Poor implementation causes “pop-in”



Kui Wu 2017, “Real-time Cloth Rendering with Fiber-level Detail”

# Level of Detail



# Conclusions:

- 3D graphics are complicated, many moving parts
- The game engine provides API and can handle things at the low level for you
- Try to use simplified representations (e.g. maps, textures, LoDs, etc.) instead of complex geometric methods, when applicable

## Final Exam

- In-Class on **Friday, May 15th, 2026 @ 4:00pm-6:00pm**
- Final-Project Demos & Presentations during **Final Week**
  - *By majority vote: May 14, May 16, or May 18, 2026*