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
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

---

High-Level 3D Graphics Pipeline

- Simplification of resources like OpenGL’s & UE3 pipeline documentation
  - 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 (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)

(Scene some other intermediate rendering passes)
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

- Geoemtric Primitives
- Concave
- Convex
- Common Shapes
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)
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

Conversion between them

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)
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)**
  - Newer API, accelerating in popularity
  - Meant to balance CPU & GPU usage
  - Much lower-level
- **Apple Metal (2015)**
  - Poor attempt to disrupt the game engine industry (deprecate OpenGL)

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)
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

---

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:
**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!

**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

Great resource for understanding different maps:
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


---

**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

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 (oooh shiny!)
- Sometimes we model **medium** (e.g. passing through water or glass)

How to actually implement PBR?

- At first, it was mostly just mixing a bunch of lighting models together, such as:
  
  ![Lighting Models](https://theovermare.com/blog/2015/02/the-journey-of-the-light-physically-based-shading/)
  
  - Ambient Diffuse
  - Ambient Specular
  - Direct Diffuse
  - Direct Specular

  = Physically Based Lighting

- 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

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++

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?

Structure of HLSL code

HLSL code has four parts:
1. Variables that get values from application
2. Input and Output structure (optional)
3. Functions
4. Techniques and passes

One HLSL file can have multiple techniques. One technique

struct VS_OUTPUT
{
    float4 pos : POSITION;
    float4 color : COLOR;
};

float4 WorldViewProj : WORLDVIEWPROJECTION;
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

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](https://www.artstation.com/artwork/mqAk0Z)
- Nice tutorials at [https://www.youtube.com/watch?v=H13BnWxY1A](https://www.youtube.com/watch?v=H13BnWxY1A) and [https://www.youtube.com/watch?v=cBlx2A3enQ](https://www.youtube.com/watch?v=cBlx2A3enQ)

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

Light Parameters & Lightmaps

- Radiance/luminosity/intensity
  - lux, candelas, lumen
- Dynamic vs static/baked lighting
Office Hours – to be updated as needs change

- Open Lab Hours: TBD
  @ AR/VR Lab (IRB 0110) on the ground level

- Office hours
  Ming: Tues/Thur after class @ IRB 5162 or by appointment <lin@umd.edu>
  Niall: Friday 12:00pm-2:00pm @ IRB 5207 or by appointment <niallw@cs.umd.edu>
  Geonsun: Wed 12:00pm-2:00pm @ IRB 5207 or by appointment <gsunlee@umd.edu>
  Jason: Tues/Thur 12:30pm-1:30pm @ AVW4176 or by appointment <jfotsopu@terpmail.umd.edu>

Rendering: Creating the Image

- 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)
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

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)**

- 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**

<table>
<thead>
<tr>
<th>Scene Setup (Geometry, transforms, etc.)</th>
<th>Scene/Model Parameter Setup (shaders, materials)</th>
<th>Rendering (Rasterization or Ray-tracing)</th>
<th>Post-processing (anti-aliasing, blur, color grading, etc.)</th>
<th>Output to Display (buffered output)</th>
</tr>
</thead>
</table>

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

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

Output: Showing the Image

Panels:
- 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)
From Image to Screen

- 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
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

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)

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!)
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”
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