3D Videogames
Università degli Studi di Milano

Rendering in 3D games
Part I: lighting environments

1

Course Plan

- lec. 1: Introduction
- lec. 2: Mathematics for 3D Games
- lec. 3: Scene Graph
- lec. 4: Game 3D Physics +
- lec. 5: Game Particle Systems
- lec. 6: Game 3D Models
- lec. 7: Game Textures
- lec. 9: Game Materials
- lec. 8: Game 3D Animations
- lec. 10: 3D Audio for 3D Games
- lec. 11: Networking for 3D Games
- lec. 12: Artificial Intelligence for 3D Games
- lec. 13: Rendering Techniques for 3D Games

bridge lectures

For a more general, deeper discussion of many of the subjects of this lecture, see the courses CG «Computer Graphics» and RTGP «Real-Time Graphics Programming»

2

One lighting equation…

repeat and sum for each light

The "ambient" term

diffuse term

specular term

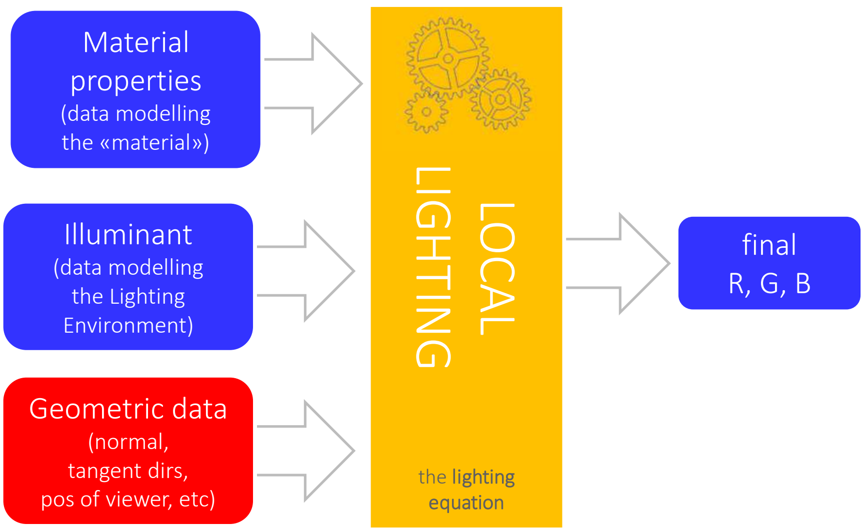
$$(\hat{n} \cdot \hat{L}) \begin{pmatrix} d_R \\ d_G \\ d_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + (\hat{n} \cdot \hat{H})^E \begin{pmatrix} s_R \\ s_G \\ s_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + \begin{pmatrix} a_R \\ a_G \\ a_B \end{pmatrix} \otimes \begin{pmatrix} A_R \\ A_G \\ A_B \end{pmatrix}$$

$\hat{H}$ = nlerp($\hat{V}$, $\hat{L}$, 0.5)
the «half-way» vector

material parameter
light parameter
3D geometric data

3

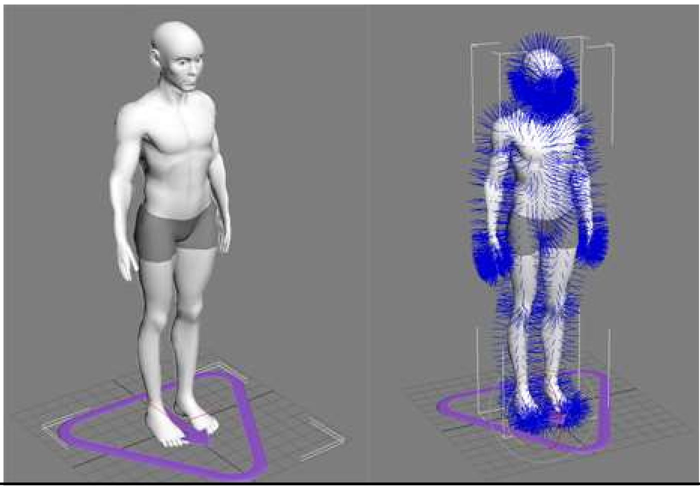
Geometric Data



4

The geometry in lighting: normals…

- Per-vertex attribute of meshes, and/or per texel (normal maps)



5

… and tangent directions, used for *anisotropic* materials

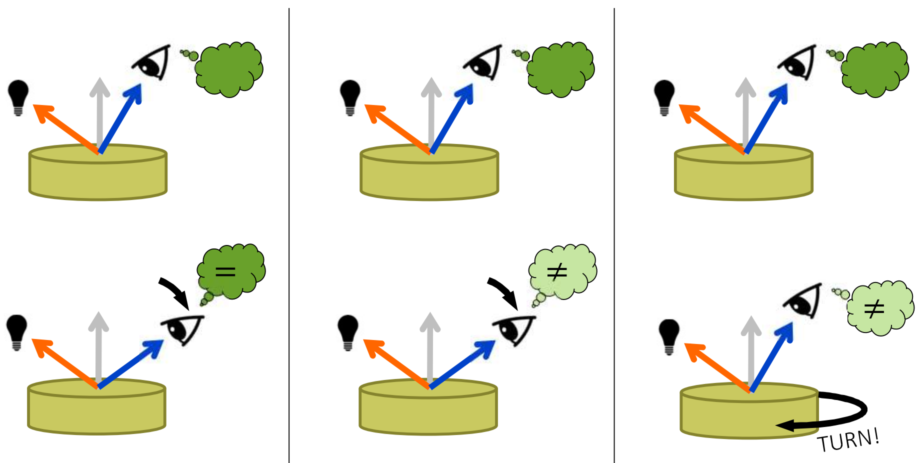
6

View-dependent and/or anisotropic materials / lighting models.

- A view-dependent lighting equation (or material) is one which uses the view direction $\hat{v}$
  - Consequence: its results cannot be backed! (why?)
  - Q: which terms of the lighting equations seen above are "view-dependent"?
  - Otherwise, it's view-independent
- An anisotropic lighting equation (or material) is one which uses the tangent directions
  - Simulates real-world materials such as: satin, velvet, fabric
  - Otherwise, it's isotropic

7

Materials / lighting models

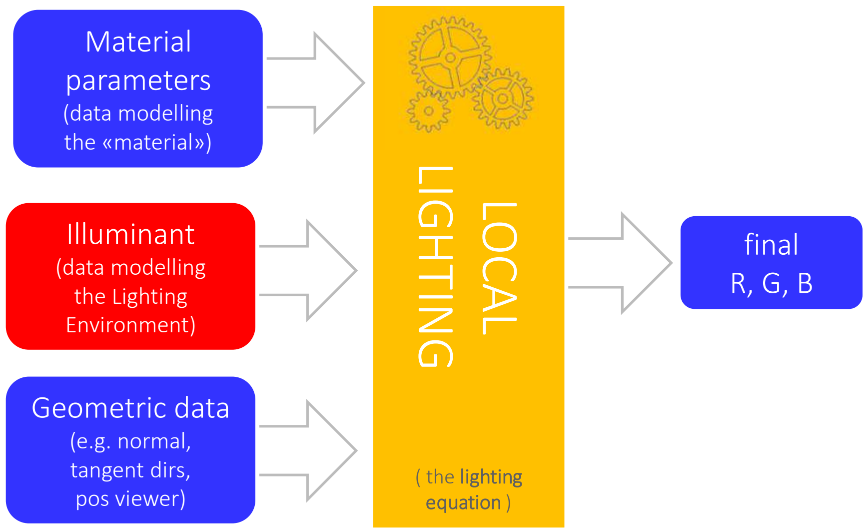


view-independent (aka Lambertian) | view-dependent (for example, Phong) | view-dependent, anisotropic

8

Marco Tarini
Università degli Studi di Milano

Next: modelling the Light environment



9

Approaches to model the light environment in 2D games

We are about to discuss three ways:

- Discrete
  - a finite set of individual light sources (including one global ambient factor for the "leftovers")
- Densely sampled
  - environment maps: textures sampling incoming light
- Basis functions
  - a spherical function stored as spherical harmonics coefficients

(They can be, and are, used jointly)

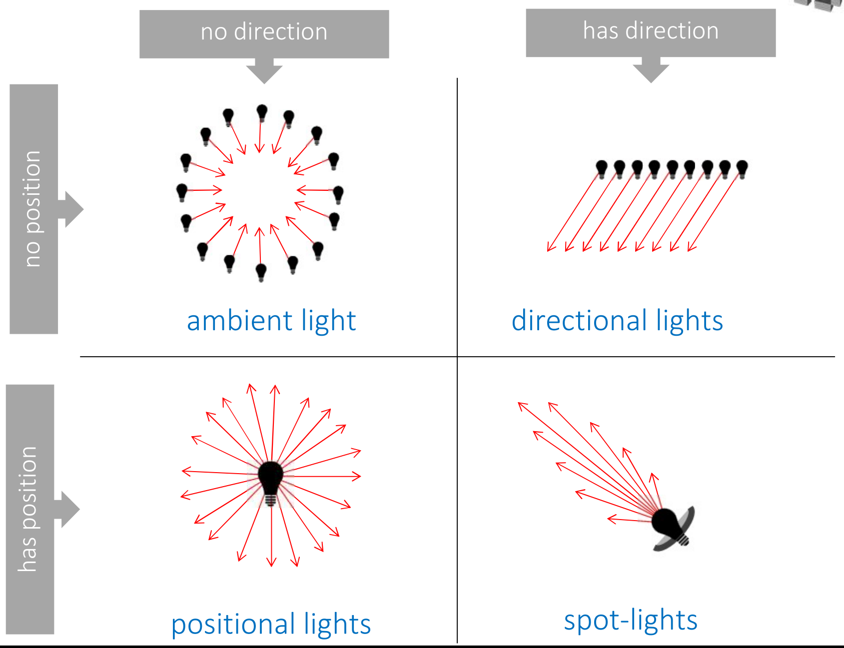
10

Discrete illumination environments: a set of individual *light sources*

- a finite set of "light sources"…
  - not too many (e.g. ≤16)
  - if more, can be assigned "priorities" to pick a subset
- each light sits in a node of the scene-graph
- each light is of one type…

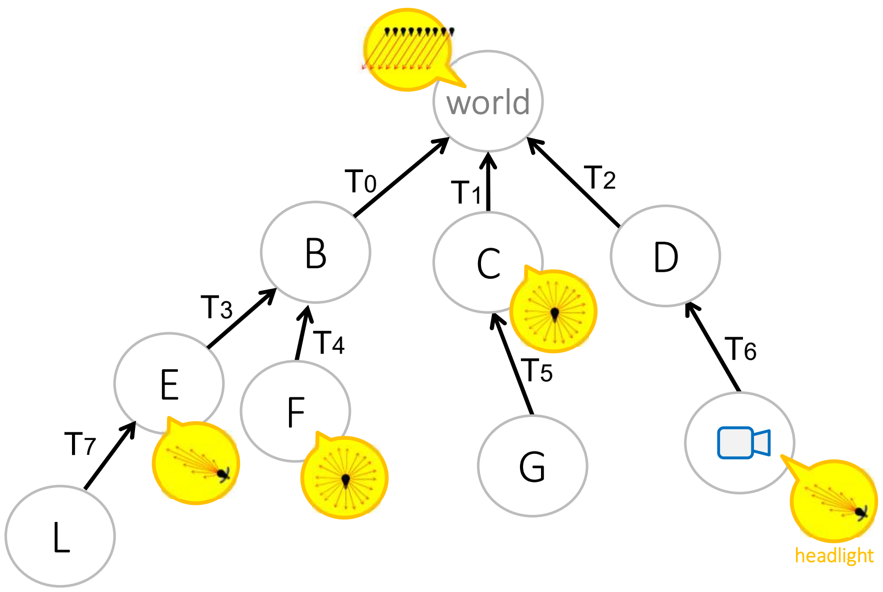
11

Types of discrete *light sources*



12

Discrete illumination environments



13

Discrete illumination environment

repeat and sum for each discrete light source

the one "ambient" light

diffuse term

specular term

$$(\hat{n}\cdot\hat{L})\begin{pmatrix}d_R\\ d_G\\ d_B\end{pmatrix}\otimes\begin{pmatrix}L_R\\ L_G\\ L_B\end{pmatrix}+(\hat{n}\cdot\hat{H})^{E}\begin{pmatrix}s_R\\ s_G\\ s_B\end{pmatrix}\otimes\begin{pmatrix}L_R\\ L_G\\ L_B\end{pmatrix}+\begin{pmatrix}a_R\\ a_G\\ a_B\end{pmatrix}\otimes\begin{pmatrix}A_R\\ A_G\\ A_B\end{pmatrix}$$

$\hat{H}$ = nlerp($\hat{V}$, $\hat{L}$, 0.5)
the «half-way» vector

material parameter
light parameter
3D geometric data

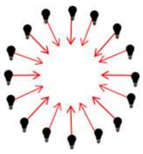
14

Types of discrete *light sources* : examples

	no direction	has direction
no position	Skylight. Cloudy day illumination. Interior light (dim) ambient light	Sunlight. Moonlight. directional lights
has position	An explosion. A wooden torch. A lightbulb. positional lights (or point-lights)	A flashlight/torchlight A spotlight Car head-lights spot-lights

15

Ambient light



- models other all minor light sources + bounces
  - light incoming "from every direction at every position"
- examples:
  - in an overcast outdoor scene: *high*
    - (dim shadows, flat looking lighting:
      every photographs' favorite for portraits!)
  - in realistic outer space: *zero*
  - in any other scenes : *something in between*
    (e.g., sunny day, or torch-lit cave)
- the lighting env includes only one (or zero) lights of this type

18

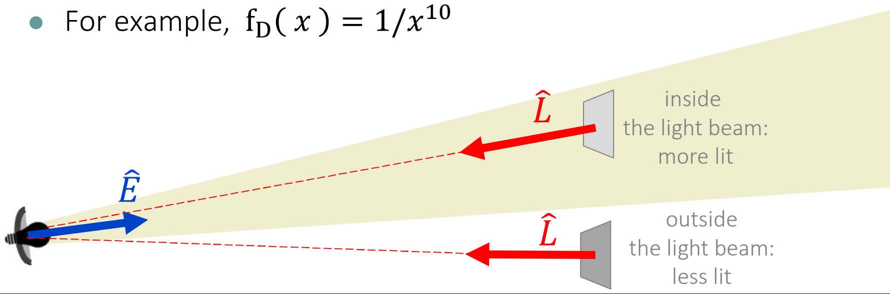
Distance fall-off functions
(for positional lights & spotlights)

- The light intensity of positional lights and spotlights can be dimmed down with distance from light-pos $\mathrm{P_L}$ to the pos of the fragment being lit $\mathrm{P_P}$, scaling it by some positional «fall-off» function
$$\mathrm{f_P}(||\mathrm{P_L} - \mathrm{P_P}||)$$
  - In the real physical world, $\mathrm{f_P}(\,d\,) = 1/d^2$
  - Other functions can be used, for example $\mathrm{f_P}(\,d\,) = 1/d$

19

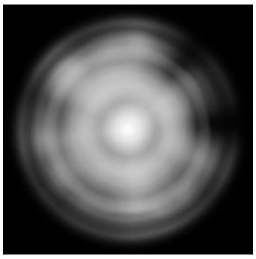
Angular fall-off functions
(for spotlights)

- For spotlights, the intensity is also dimmed down by an «angular fall-off» function, when the direction of the light emission $\hat{E}$ mismatches the light direction $\hat{L}$, scaling it by some function
$$\mathrm{f_D}\left(-\hat{E} \cdot \hat{L}\right)$$
  - For example, $\mathrm{f_D}(\,x\,) = 1/x^{10}$



20

Spot-lights: they can use a "cookie texture"



As an alternative to use angular fall-off functions

21

In the lighting equation…

repeat and sum for each discrete light source

the one "ambient" light

diffuse term

specular term

$$(\hat{n} \cdot \hat{L}) \begin{pmatrix} d_R \\ d_G \\ d_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + (\hat{n} \cdot \hat{H})^E \begin{pmatrix} s_R \\ s_G \\ s_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + \begin{pmatrix} a_R \\ a_G \\ a_B \end{pmatrix} \otimes \begin{pmatrix} A_R \\ A_G \\ A_B \end{pmatrix}$$

Light incoming direction
For directional lights: just a constant.
For a point- or spot-light: $\hat{L} = \frac{P_L - P_P}{\|P_L - P_P\|}$

pos of light — pos of lit point

Light RGB intensity
For a dir. light: a consts.
For point or spot-lights: attenuated by falloff functions (of angle, dist)

(yellow highlight) = parameter of the light

22

Types of discrete lights (a summary)

	Ambient light	Directional light	Positional light	Spotlight
geometry	*(nothing)* *(assumed at infinite)*	Direction (versor) *(assumed at infinite)*	Position (point)	Position (point) & Direction (versor)
can be dimmed by	-	-	Falloff function	Falloff function Angular falloff function "Cookie" texture
can be blocked by	Ambient Occlusion either baked (per-vertex or per-texel) or dynamically computed (see SSAO later)	Cast shadows (usually) dynamically computed (see shadow-map technique later)		
how many	0-1	0-N	0-N	0-N
parameters	Color/Intensity (RGB value) Priority?			

23

In which space to compute the lighting?

repeat for each light source

$$\underbrace{(\hat{n}\cdot\hat{L})\begin{pmatrix}d_R\\d_G\\d_B\end{pmatrix}\otimes\begin{pmatrix}L_R\\L_G\\L_B\end{pmatrix}}_{\text{diffuse term}} + \underbrace{(\hat{n}\cdot\hat{H})^E\begin{pmatrix}s_R\\s_G\\s_B\end{pmatrix}\otimes\begin{pmatrix}L_R\\L_G\\L_B\end{pmatrix}}_{\text{specular term}} + \underbrace{\begin{pmatrix}a_R\\a_G\\a_B\end{pmatrix}\otimes\begin{pmatrix}A_R\\A_G\\A_B\end{pmatrix}}_{\text{ambient term}} + \underbrace{\begin{pmatrix}e_R\\e_G\\e_B\end{pmatrix}}_{\text{emission term}}$$

$$\hat{L} = \frac{P_L - P_P}{\|P_L - P_P\|}$$

$\hat{H} =$ nlerp($\hat{V}$, $\hat{L}$, 0.5)

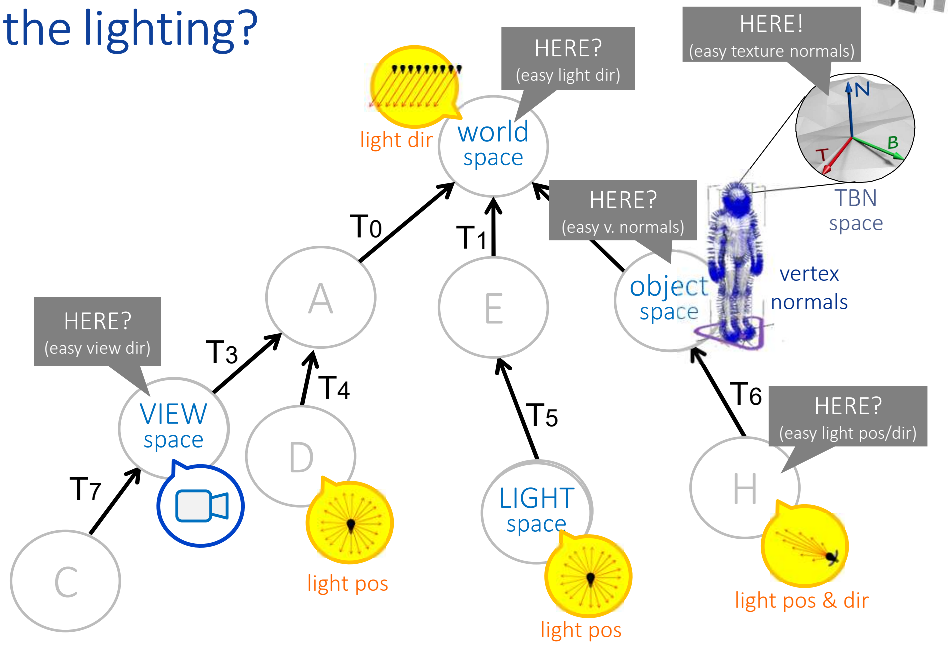
the «half-way» vector

3D Point / Vector / Versor

Q : in which space to express them (and the others like them)?
A: whichever!
As long as it's the *same space*

24

In which space to compute the lighting?



25

In which space to compute the lighting?

- All versors that used in the lighting equation must be expressed in the same space
  - view direction, light directions, half-way vector, normals, tangent dirs… (↑ for anisotropic materials)
- Choice: which space to use?
  - View space? (the space of the camera)
  - World space?
  - Local object space? (the space of the object currently being rendered)
- With normal maps, usually the most efficient solution is:
  - Use the same space the normals are expressed
  - For normal stored as attribute: the Local Space (aka Object Space)
  - For Tangent Space normal maps: in the the TBN space. Then…
  - …all other versors must be transformed into this space, *per vertex!*
  - …the normals accessed from the texture can be used right away, *per pixel!*
  - This minimizes the amount of transformations needed

26

Discrete illumination environments Summary

- Pros:
  - simple to position / reorient individual light sources
    - both at design phase, or dynamically (at game exec)
  - good model of illuminants, such as:
    - explosions (positional lights)
    - car lights (spot-lights lights)
    - sun direction (directional light)
  - relatively easy to compute (hard, soft) shadows for them
- Cons:
  - each light source requires extra processing ... for each pixel!
    - therefore: hard limit on their number. Prioritize
    - therefore: are often given a (physically unjustified) radius of effect
  - they don't model well:
    - area light sources (e.g., from back-lit clouds)
    - reflections on metal objects

main illuminants of the scene!

see shadow map later

27

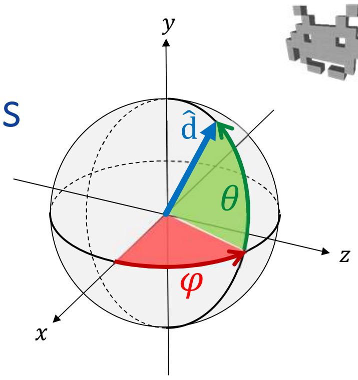
Densely sampled illumination environments

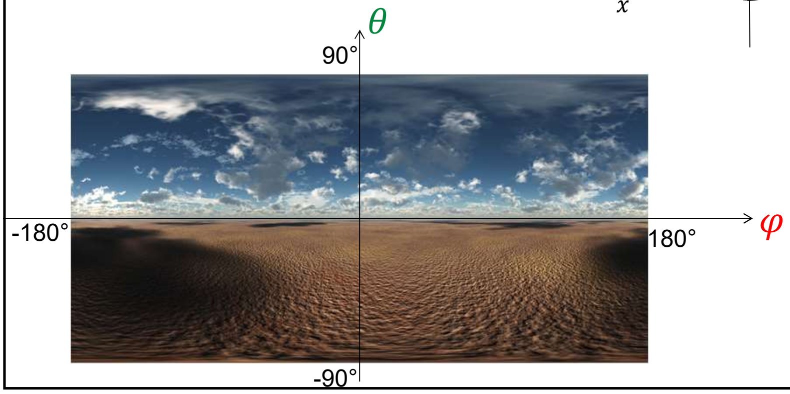
- A light intensity / color from each direction $\hat{\mathrm{d}}$
- Asset to store that:
  "Environment map" texture

28

Densely sampled illumination environments

- Latitude/longitude format (of a unit vector $\hat{d}$)





29

Densely sampled illumination environments

- Aka “sky-map” texture
  - when it’s only / predominantly the sky to be featured
  - doubles as textures for “sky boxes”

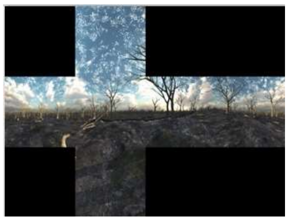
30

Densely sampled illumination environments

- Environment map: (asset)
  a texture with a texel $\mathbf{t}$ for each direction $\hat{\mathrm{d}}$ (unit vector)
  - $\mathbf{t}$ stores the intensity/color of the light coming from direction $\hat{\mathrm{d}}$
- Q: how to determine u, v position of $\mathbf{t}$ for a given $\hat{\mathrm{d}}$?
  - i.e. how to parametrize (flatten) the unit sphere
- Different answers are possible…


latitude/longitude format


mirror sphere format


cube-map format (ad-hoc HW support!)

31

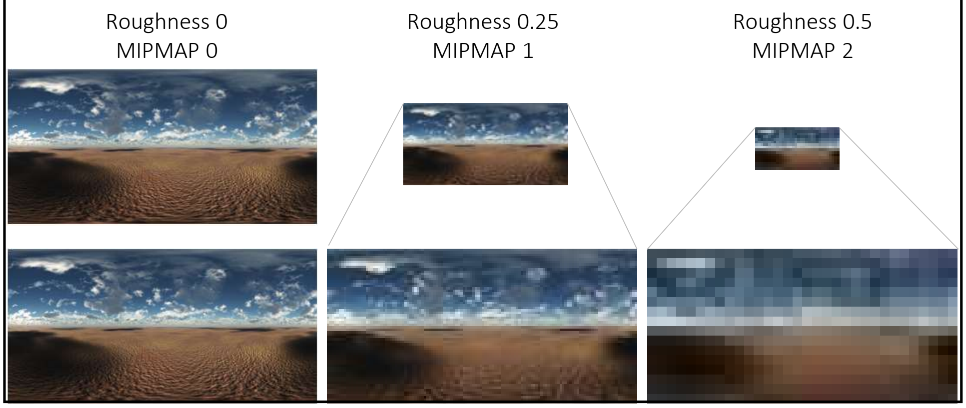
Environment map (asset)

- A texture with a texel $\mathbf{t}$ for each direction $\hat{\mathrm{d}}$
  - $\mathbf{t}$ stores the light coming from direction $\hat{\mathrm{d}}$
  - useful to compute reflections on (curved) metallic objects
  - often HDR (see later)
- Pro: realistic, complex, detailed, hi-freq, light env
  - best for mirroring materials (such as metal, glass, water)
- Pro: can be captured from reality
  - see "mat-cap"
- Con: expensive to update for dynamic scenes
  - no prob, for static environments only
- Con: assume far away illuminants
  - Not accurate for close illuminant

32

Environment map (asset): uses

1. Reflection mapping
   - metallic objects
   - material roughness → mipmap level!



33

Environment map (asset): uses

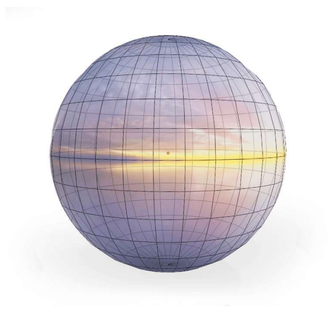
1. Reflection mapping
   - metallic objects
   - material roughness → mipmap level!

2. More generally,
   description of the lighting env
   - for lighting computation

3. Coverage of the background
   - e.g., as a texture covering the
     3D “skybox” / “skydome” mesh

34

Skydome mesh



Its transform:

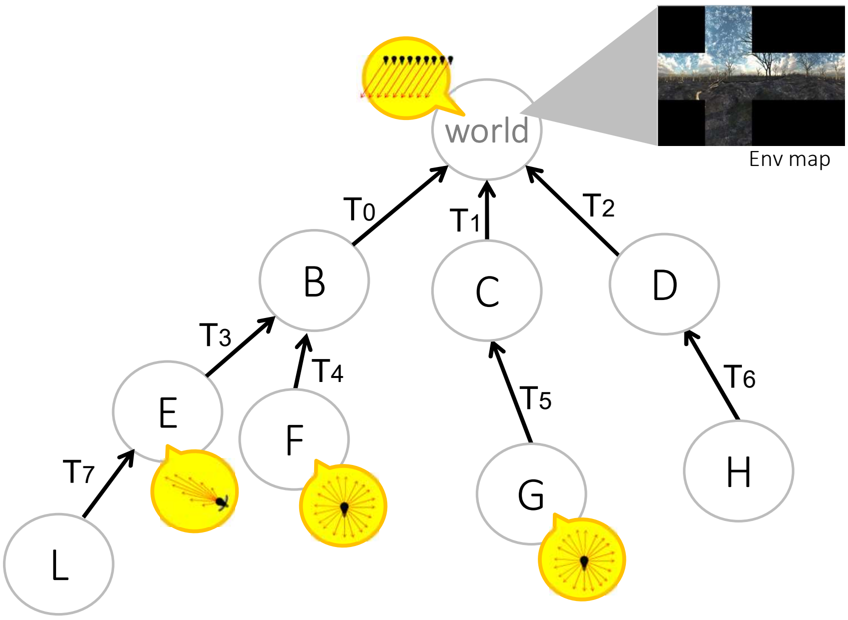
- Centered at camera node
- Oriented as world node

Textured with:

- The environment map

35

Lighting env in the scene graph



36

Marco Tarini
Università degli Studi di Milano

Light environments: using Basis Functions

- Lighting environment: a *continuous* function $f : \Omega \rightarrow \mathbb{R}$
- $f(\hat{v})$ = amount of (rgb) light coming from direction $\hat{v}$
- Store f through basis functions

set of all unit vectors (i.e., surface of the unit sphere)

or R^3 if RGB colored light

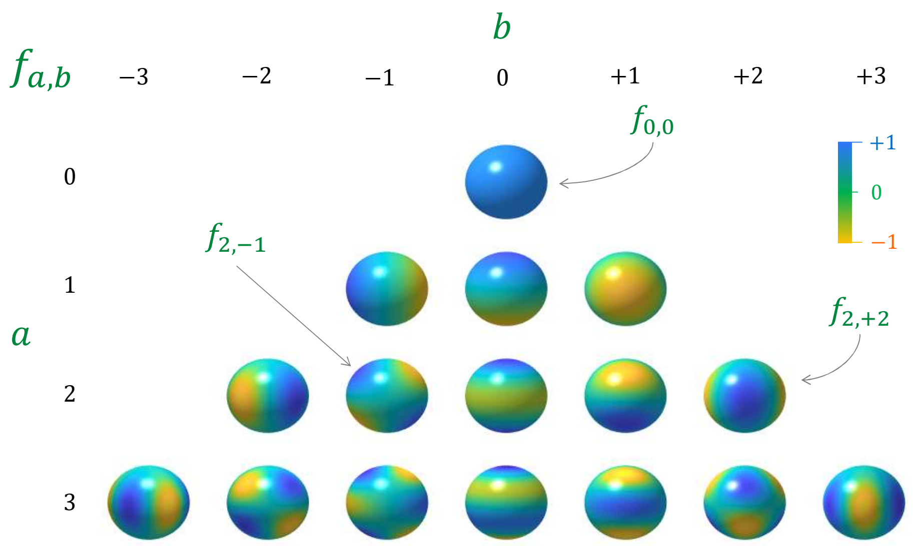
fixed spherical "basis" functions (always the same ones)

$$f(\hat{v}) \cong a_{0,0} \cdot f_{0,0}(\hat{v}) + a_{1,-1} \cdot f_{1,-1}(\hat{v}) + a_{1,0} \cdot f_{1,0}(\hat{v}) + a_{1,+1} \cdot f_{1,+1}(\hat{v}) + \cdots$$

a few scalar values to be stored, in order to represent (an approx. of) f

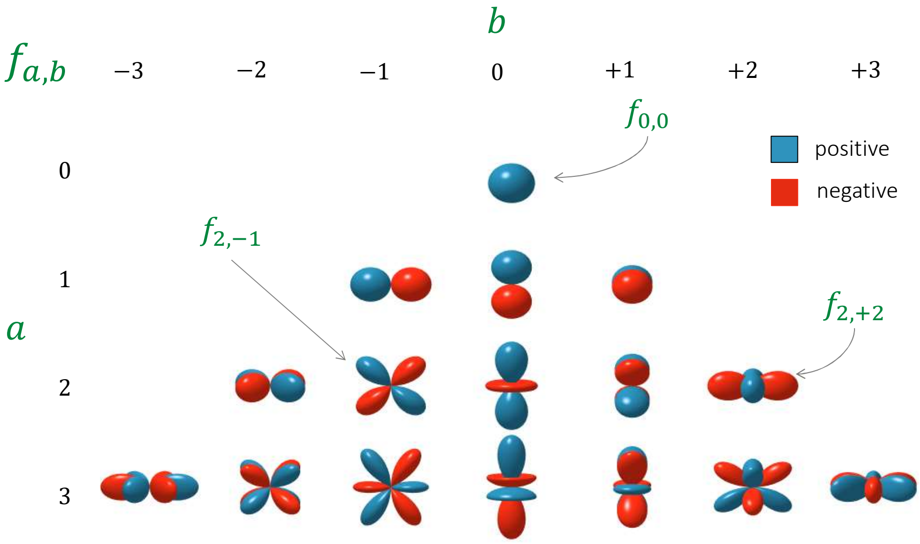
37

Spherical Harmonics (SpH): a set of functions



39

Spherical Harmonics (SpH): a set of functions (a different visualization)



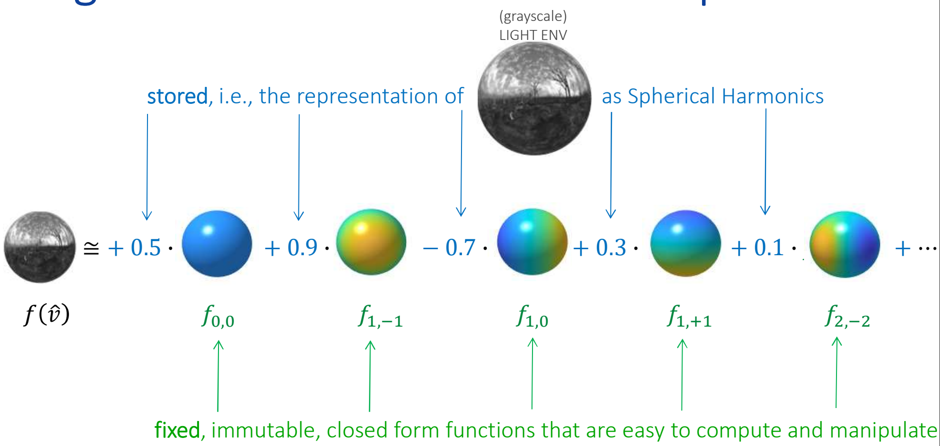
40

Spherical Harmonics (SpH): a good choice for the basis functions

- Spherical Harmonics is a good set of basis functions for spherical functions
- Each function in the set has two indices a, b (the degree)
  - $f_{a,b}(\hat{v})$ with $a \geq 0\,, \quad -a \leq b \leq +a$
  - $f_{0,0}(\hat{v}) = 1$ a constant function (so, scalar $a_{0,0}$ represent the *total amount* of light)
  - all other basis function sum up to 0 (i.e., their integral over Ω is zero) — so, they control the distribution not the *quantity*, of light
  - they are designed to have useful mathematical properties (e.g., orthogonality – the integral of the product of any two is 0)
  - all SpH functions are easy to compute, e.g. integrate, etc

41

Light probes: Light environment stored with SpH



f is stored as $(+0.5, +0.9, -0.7, +0.3, 0.1, \ldots)$

(if it's a colored Light Env, this is repeated for each R,G,B channel)

42

Light probes: Light environment stored with SpH

- Spherical Harmonics (SPH) in brief:
  - store Illumination Env as a small number (4,9,16…) of scalar weights of as many fixed spherical basis functions.
- Pros:
  - very compact representation
  - it models continuous functions well: good for smooth lighting environments
  - it allows for efficient computation of the Lighting equation
  - it's easy to interpolate between light envs!
- Cons:
  - continuous functions *ONLY*
    - Not good for hi-freq details: for example, no hard lights
    - not sudden variations (unless very many coefficient used)
- Good for soft light env

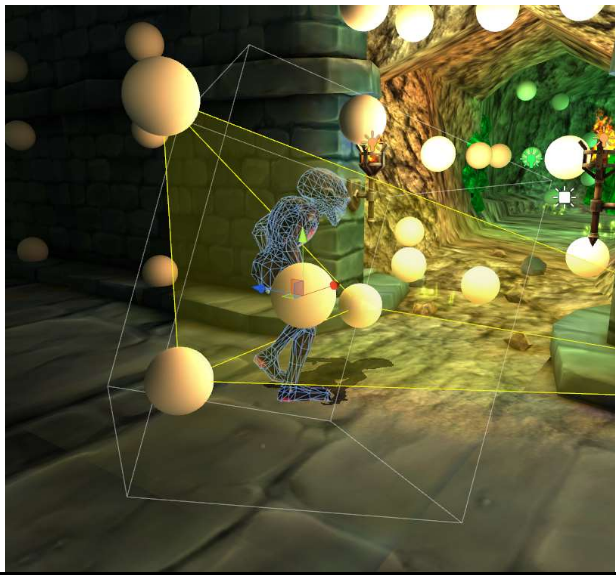
43

Light probes (position-dependent lighting env)

- A light probe == a (precomputed) lighting env. to be used around a given 3D position of the scene
- Light Probe lighting:
  - preprocessing: disseminate the scene with light probes
    - Store them as… low-res environment maps
    - …or, with SPH (the standard solution)
  - at rendering time, for an object currently in pos (xyz), use an interpolation of near-by "light probes"
    - note: two (or more) SPH function can be interpolated!
    - easy: just interpolate the weights

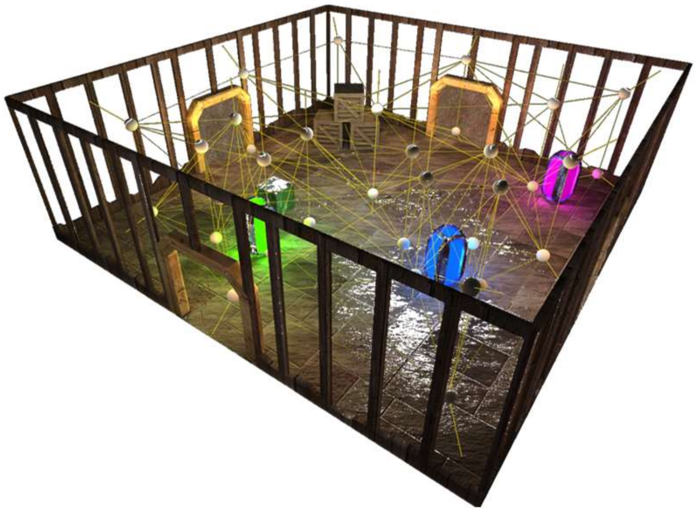
44

Light probes (position-dependent lighting env)

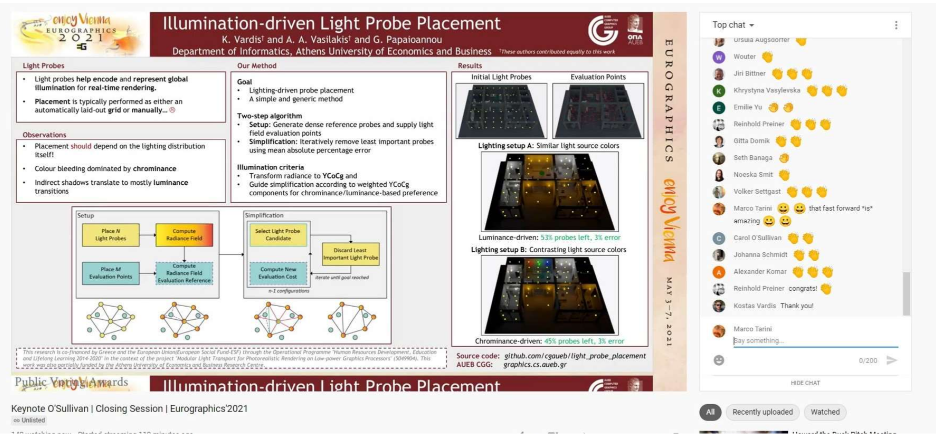


45

Light probes
(position-dependent lighting env)



46



47

Rendering in 3D games

Part II: Common CG techniques in games

50

Part II

- Basics of GPU-based rendering
  - a brief summary of rasterization based rendering
  - programmable parts of the pipeline
  - depth-maps
  - double buffering
- Rendering techniques & tricks used in games
  - Multi-pass techniques in general
  - Deferred shading
  - Screen space techniques in general
  - A summary of a few common CG techniques

51

Rendering task for in 3D games: overview

- Real time
  - (20 or) 30 or 60 FPS
- Hardware (GPU) based
  - pipelined, stream processing
- therefore: one class of algorithms (hardwired)
  - rasterization based algorithm
  - recent trend: switch to ray-tracing algorithms?
- Complexity:
  - Linear with # of primitives
  - Linear with # of pixels

52

High-level view of mesh rendering

To render a mesh:

- load in GPU RAM:
  - ✓ Geometry + Attributes
  - ✓ Connectivity
  - ✓ Textures
  - ✓ Vertex + Fragment Shaders
  - ✓ Global Material Parameters
  - ✓ Rendering Settings
- issue the Draw-call

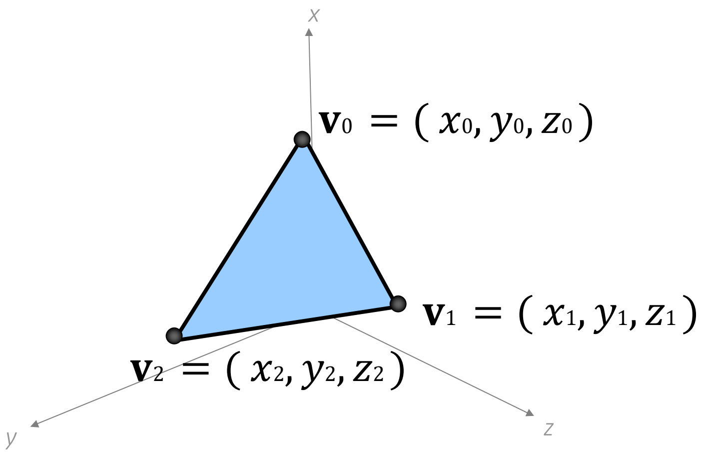
THE MESH ASSET

THE MATERIAL ASSET

For this lecture, we need go lower level (a bit)

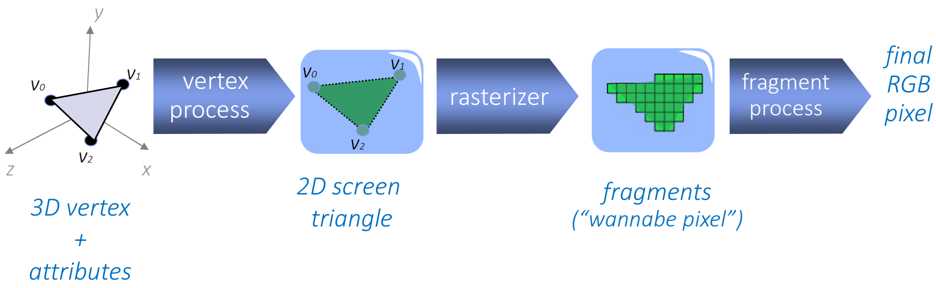
53

Rendering of a mesh = rasterization of all its triangles



56

GPU pipeline – a simplified conceptual version



57

Rasterization based rendering: steps (remarks 1/2)

- Vertex processor: (per vertex)
  - Input: vertex data (position + initial attributes)
  - Output: a final screen position, and other (refined) attributes
- Rasterizer: (per triangle)
  - Input: a triplet of processed vertex (with attributes)
  - Output: many "fragment", one for each pixel covered by the triangle, each with interpolated attributes
- Fragment shader: (per fragment)
  - Input: a fragment (with attributes)
  - Output: a final rgb color (plus: an alpha value, plus: a depth value)
- Output combiner: (per fragment)
  - Writes the rgb color on the screen buffer
  - Overwrites, blends, or preserves the old value

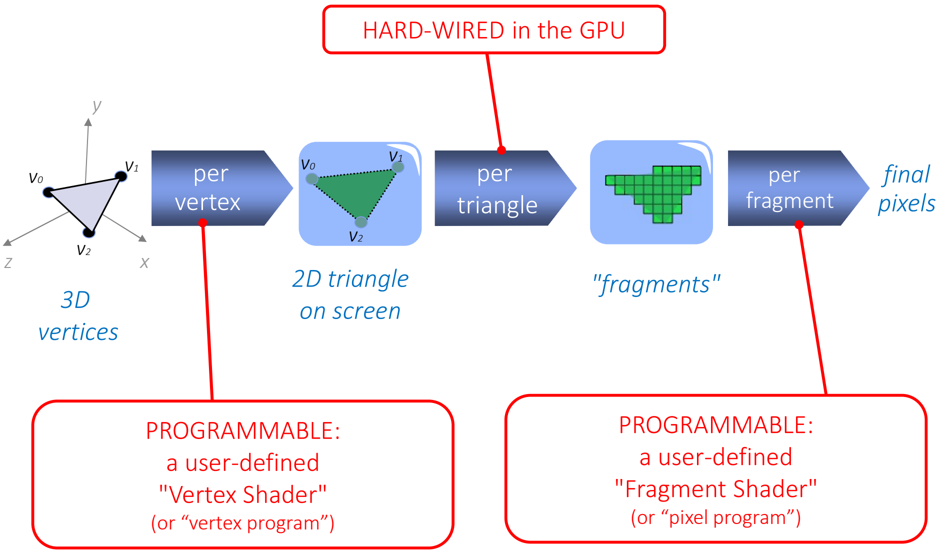
62

Rasterization based rendering: steps (remarks 2/2)

- It's a pipelined architecture: every step works in parallel with all others
  - E.g., while fragment are processed, the next triangle is being rasterized, and the next vertices are processed
- It's a SIMD architecture: Every step does the same processing on several inputs, producing several output, all in parallel,
  - E.g., several fragments are processed at the same time (each one independently from the others)
  - E.g., same for vertices

63

Rasterization-Based Rendering



65

Rasterization based rendering: what is computed in each step

the Vertex Shader

- Per vertex:
  - shape-blending - interpolation of the current morphs
  - skinning - transform from rest pose to current pose (blending matrices)
  - projection - transform from object space to screen space

hard wired

- Per triangle: (rasterizer)
  - back-face culling discard of back-facing triangles
  - rasterization creation of fragments
  - interpolation of per-vertex attributes ←*nota bene!*

the Fragment Shader

- Per fragment:
  - texturing: accessing textures (including color, normal, alpha maps)
  - lighting: from normal + lights + material to RGB
  - alpha-kill: discard of (almost) fully transparent fragments
  - fog effect: pixels distant from the camera are blended with a "fog" color

hard wired

- Per fragment: (output combiner)
  - depth-test: occluded pixels are removed (often anticipated for performance)
  - alpha-blend: semi-transparent fragments are mixed with background

66

GPU pipeline – bottlenecks
(remarks and terminology)

- Like in any pipeline, the process goes *as slow as its slowest stage*
  - i.e., the «bottleneck» of the pipeline determines the total speed
  - Any other stage is idle for part of the time (which is always a waste)
    - stages before the bottleneck are «chocked»
      (they cannot produce output because next stage is not ready)
    - stages after it are «starved» (they wait for input from previous stage)
- Bottleneck terminology: (in CG)
  - If the bottleneck is per vertex, the app is goemetry-limited
    («it cannot process geometry fast enough»)
  - If the bottleneck is per fragment, the app is fill-limited
    («it cannot fill the screen buffer with pixel fast enough»)

MORE COMMON CASE, FOR GAMES

- Performaces (rendering FPS) of a game only impoves
  if computational load is removed from the bottleneck phase
  Examples:
  - using all meshes at LOD 2 instead of 0 does not help a fill-limited app
  - reducing the resolution of the screen does not help a geometry-limited app
  - using a simpler lighting model does not help a geometry-limited app

67

In many game engines, shaders are part of the "material asset"

To render a mesh:

- load (in GPU RAM):
  - ✓ Geometry + Attributes
  - ✓ Connectivity
  - ✓ Textures
  - ✓ Vertex + Fragment Shaders
  - ✓ Global Material Parameters
  - ✓ Rendering Settings
- issue the Draw-call

THE MESH ASSET (Geometry + Attributes, Connectivity)

THE MATERIAL ASSET (Textures, Vertex + Fragment Shaders, Global Material Parameters, Rendering Settings)

68

Programming languages for writing shaders

- High level:
  - HLSL (High Level Shader Language, Direct3D, Microsoft)
  - GLSL (OpenGL Shading Language)
  - CG (C for Graphics, Nvidia)
  - PSSL (PlayStation, Sony)
  - MSL (Metal, Apple)
- Low level:
  - ARB Shader Program
    (the "assembler" of GPU – now deprecated)

69

Depth buffer (or Z-buffer) (or depth-map)

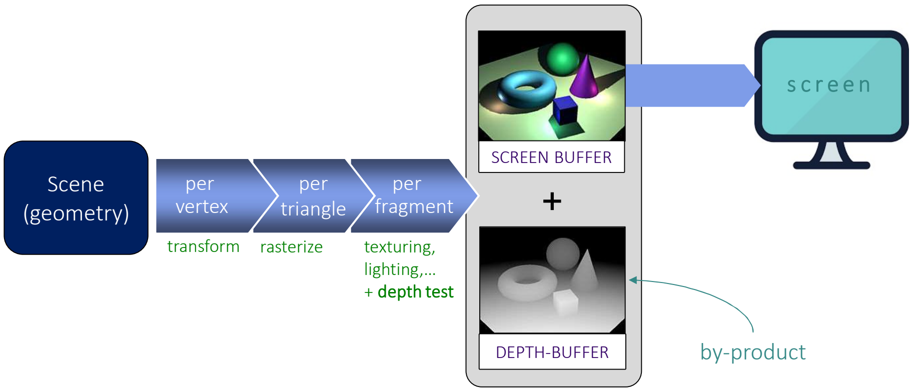
- Any rendering producing a screen-buffer …
  - which is sent to the screen
- …also produces a depth-buffer
  - as a by-product!
  - not set to the screen: it's an "offline" buffer
  - it's used during the rendering to determine occlusions and remove "hidden surfaces"
    (i.e. make what is behind something else is not seen, because it's covered by that something)
  - see computer graphics course for more details
- many rendering algorithms exploit the depth-buffer
  - for different uses
  - for each pixel on the screen, we have not only its RGB value, but its depth value (a scalar from 0 – close to the camera, to 1 – far from the camera)

(screen-buffer:) a 2D array of RGB values of some resolution

(depth-buffer:) a 2D array of depth values (scalars in 0 to 1) of the same resolution

70

basics: Depth buffer



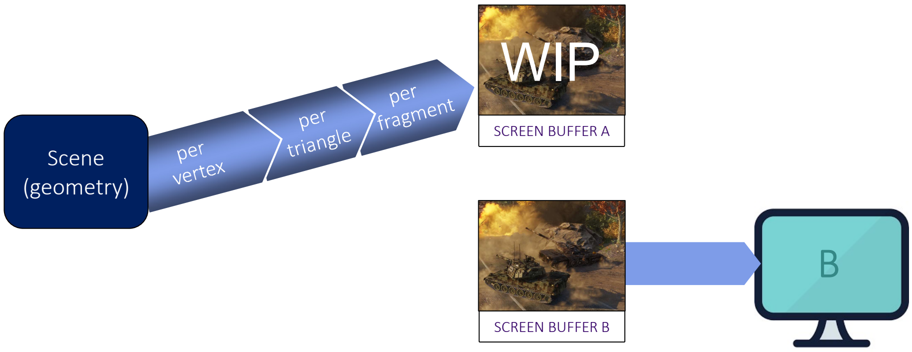
71

basics: Double Buffering

- To render a scene, all meshes are rendered succession
  - Filling the screen buffer
- Double-buffering is a basic technique to prevent any incomplete buffer to ever reach the screen
  - E.g., a rendering where some of the meshes is still not rendered
- How it works:
  - We have two RGB buffers: the front-buffer and the back-buffer
  - The front buffer shows the last complete rendering and is the one the screen shows
  - The back buffer is filled by the renderings, but it is not shown (it’s yet another example of “off-screen buffer”)
  - Screen Swap: When the back buffer is ready, the two buffer are swapped (instantaneously)
  - Info about variants: look up what “V-sync” means in 3D games settings
  - Observation: the depth-buffer needs not be doubled

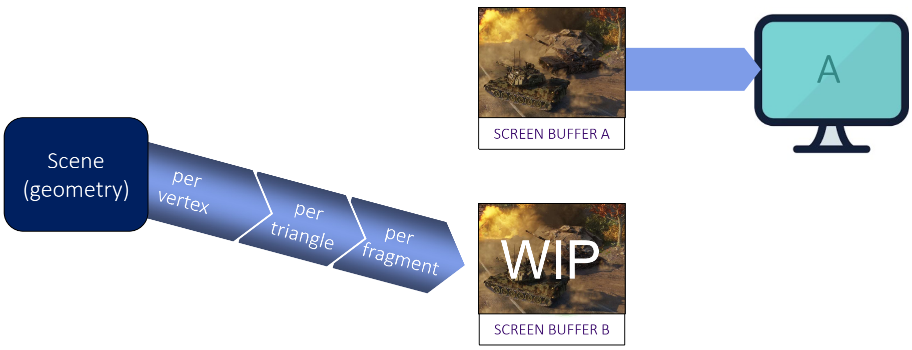
72

basics: Double Buffering



73

basics: Double Buffering



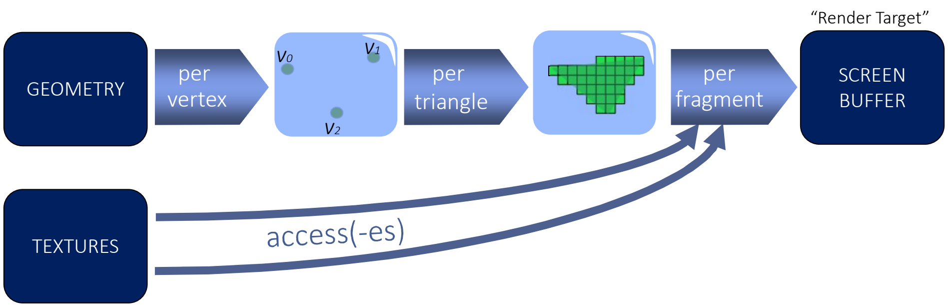
74

Off-screen buffers

- The rendering produces a screen buffer (2D array of RGB pixel) that is sent to the screen and is made visible to the player
- A buffers that is used internally but and not sent to the screen is called an off-screen buffer
  - The depth buffer (2D array of depth values)
  - The back-screen buffer (double buffering techniques)
- Many rendering techniques are based on producing then using an off-screen buffer

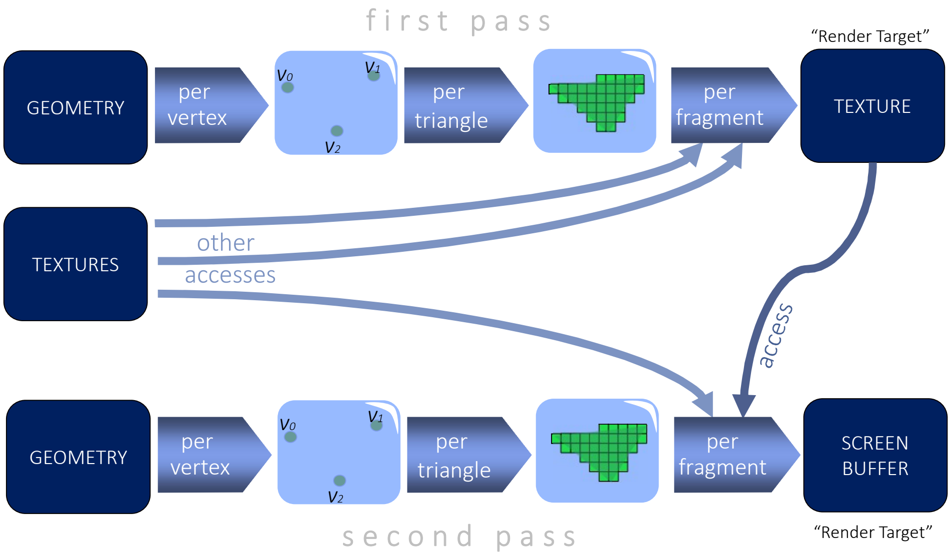
75

Texture access: it's in the per fragment process



77

basics: Render to Texture
(multi-pass rendering)



79

Multipass rendering techniques
(a wide class of rendering techniques)

- 1st pass: fill an internal 2D buffer
  - i.e., an "off-screen" buffer (a buffer never shown to the user)
  - it's the output of this rendering, i.e. its "render target"
  - by default, the render target is the "screen buffer" (the buffer shown to the screen), but not in this case
  - this mechanism is aka "render to texture"
- 2nd pass: fill the final screen buffer
  - using the just-computed off-screen buffer as a 2D texture
- Note: good for GPU because…
  - the off-screen buffer is either only write-only (1st pass) or read-only (2nd pass). Never both!
  - the off-screen buffer is constructed and used in GPU RAM. No expensive swap of memory between CPU and GPU!

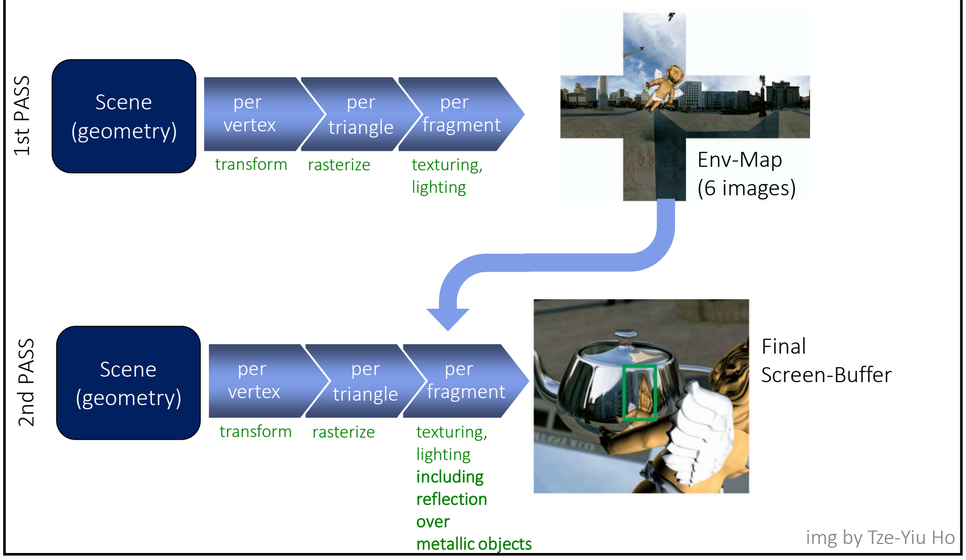
80

Screen-Space techniques (in general)
(a class of multi-pass rendering techniques)

- 1^{st} pass:
  - Render the scene from the same point of view as the final scene
  - Produce: final color buffer, plus a z-buffer (and/or other auxiliary buffers)
- 2^{nd} pass:
  - render just one single “full screen” rectangle
  - (it fills the entire screens with two triangles)
  - for each produced fragment: apply 2D effects to the buffer
- Notes:
  - Basically, it’s a way to apply “post-production” 2D image filters after the rendering.
  - Many of the techniques in these slides are in this category

82

Example: metallic reflections of *dynamic* scenes



img by Tze-Yiu Ho

84

Main rendering algorithms: two classes of approaches

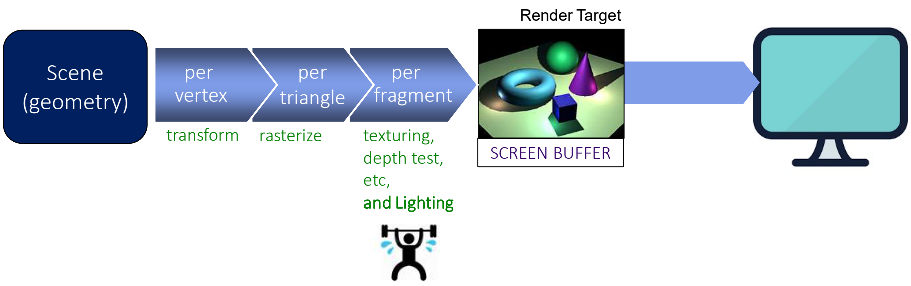
- Forward rendering
- Deferred shading ← aka Deferred lighting (actually, a variation) / aka Deferred rendering (inappropriate?)

- Which approach to use?
  - Both are employed by games
  - Basilar choice! Implementation of all other rendering algorithms changes accordingly.

85

Main rendering algorithms: two classes of approaches

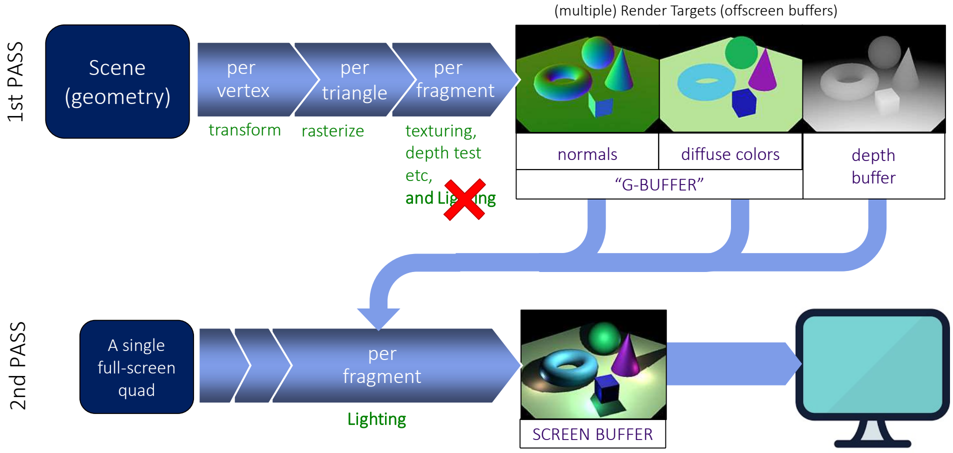
- Forward rendering



86

Main rendering algorithms: two classes of approaches

- Deferred shading ← aka Deferred lighting (actually, a variation)
  aka Deferred rendering (inappropriate?)



87

Deferred shading

- Advantage:
  lighting is computed only actually visible pixels
  - it's a huge saving if large depth complexity (aka overdraw) and/or lighting complexity – both common in 3D games
- Disadvantage:
  needs a separate buffer for every material parameter (or, sometimes, a material index)
  - Normal buffer
  - Depth buffer
  - Base color buffer
- Limits the range of materials?
- Disadvantage: not good for semi-transparencies

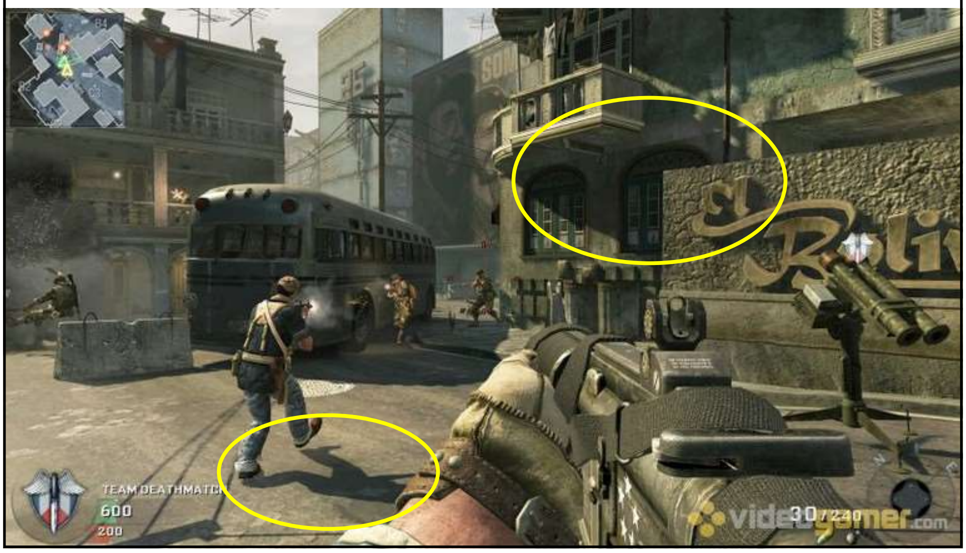
88

Some rendering techniques popular in games (we will see a few of them)

- Shadowing
  - shadow mapping ← with PCF
  - Screen Space Ambient Occlusion ← SSAO
- Camera lens effects
  - Flares
  - limited Depth Of Field ← DoF
- Motion Blur
- High Dynamic Range ← HDR
- Non-Photorealistic Rendering ← NPR
  - e.g., cell shading:
  - 1. contours
  - 2. lighting quantization
- Texture-for-geometry
  - Bump-mapping
  - Parallax mapping

89

Shadow mapping



90

Shadow-mapping in a nutshell (a multi-pass technique for shadows)

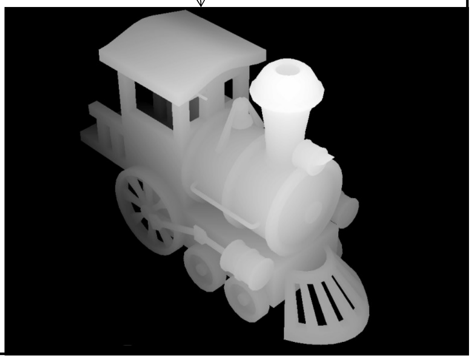
1st pass:

- camera in light position
- render all light blockers
- produce a depth buffer *only* (known as the shadow map)
- (repeat for each discrete light casting a shadow)

2nd pass:

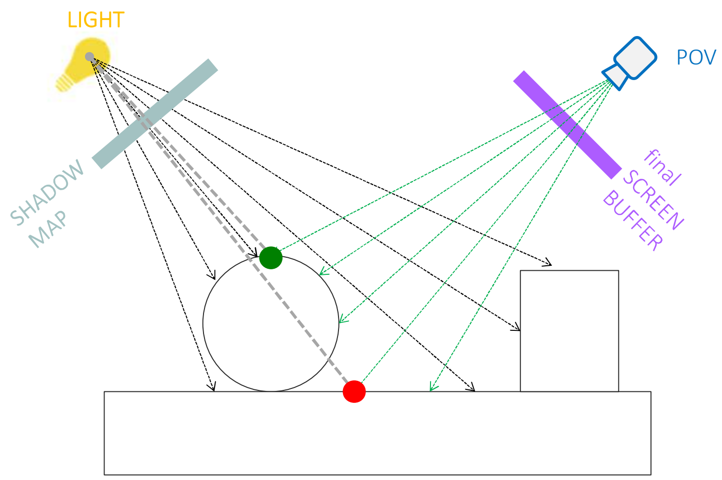
- camera in final position
- for each fragment, access the shadow-map, determine if that if fragment is visible by light (or not)
- If not visible, negate contribution of that discrete light source

- Result:
  - Blockers cast a shadow



92

Shadow-mapping concept



93

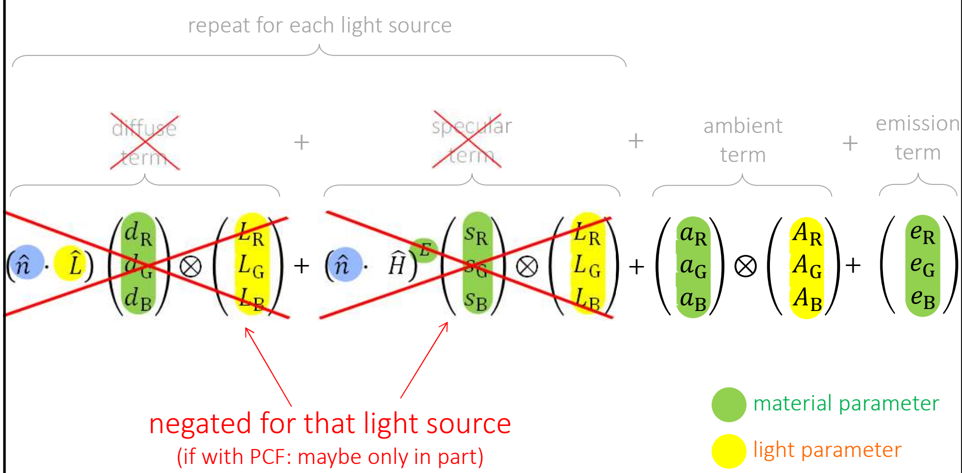
Shadow mapping: issues

- Rendering shadow-map:
  - Must be redone every time object move
  - can be baked once and for all, for static objects only
  - (jet another reason to label static objects!)
- Shadow-map resolution:
  - it matters! aliasing effects
  - remedies: PCF, multi-res shadow-map

optional topics (no exam)

94

Shadow Mapping: effect of being in shadow



95

Shadow Mapping: effect of being in shadow

- Negates (zeroes) the light term of that (discrete) light-source (positional, directional, or spot- lights)
- Observe: the other lights are unaffected:
  - Other (non shadowed) positional / directional lights
  - Any ambient light
  - Also, the emission factor (if present)

96

Two ways to compute AO: static AO versus SSAO

- Static Ambient Occlusion (or Baked AO)
  - Baked in preprocessing on each mesh, in Object Space
  - Stored as a per-vertex attribute OR a texture (called "AO-map", or "light-map")
  - Pro: accurate & cheap (during rendering)
  - Con: static! Doesn't reflect current pos of the objects in the scene
- Screen Space Ambient Occlusion (SSAO)
  - It's a screen-space technique:
  - 1^{st} pass: compute depth map (maybe normals too)
  - 2^{nd} pass: compute AO map from the above (AO factor of each pixel, depends on neighboring depth values)
  - Final pass: use AO per-pixel from pass 2
  - Pro: dynamic! Reflect current position of objects in the scene
  - Con: less accurate
- The two can be combined!

98

Screen Space Ambient Occlusion (SSAO)

SSAO only

99

Ambient occlusion (AO)

- Cast shadows (computed by shadow-maps) negate the light coming from discrete light sources
- "Ambient occlusion", negates (occludes) the "ambient" component of lighting, instead
- Idea:
  - the AO is a factor (between 0 and 1) for each surface point
  - AO factor multiplies the ambient component for that point
  - Intuitively, for a point **p**, its AO factor is a measure of how much **p** is exposed in the open
    - **p** is well exposed: AO ≈ 1.0
    - **p** is hidden, e.g. it is in the bottom of a crack: AO ≈ 0.0
  - Exact definition - not in this course. But keep in mind:
    - (1) it is an approximation
    - (2) it is a purely geometrical computation

100

Ambient occlusion: effects

repeat for each light source

diffuse term + specular term + ~~ambient term~~ + emission term

$$(\hat{n} \cdot \hat{L}) \begin{pmatrix} d_R \\ d_G \\ d_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + (\hat{n} \cdot \hat{H})^E \begin{pmatrix} s_R \\ s_G \\ s_B \end{pmatrix} \otimes \begin{pmatrix} L_R \\ L_G \\ L_B \end{pmatrix} + \begin{pmatrix} a_R \\ a_G \\ a_B \end{pmatrix} \otimes \begin{pmatrix} A_R \\ A_G \\ A_B \end{pmatrix} + \begin{pmatrix} e_R \\ e_G \\ e_B \end{pmatrix}$$

negates some % of this

material parameter
light parameter
geometry

104

(limited) Depth of Field

105

(limited) Depth of Field in a nutshell

- Screen space technique:
- 1st pass: standard rendering, producing
  - RGB image (kept off screen)
  - depth-buffer (as usual)
- 2nd pass:
  - pixel inside of focus range? Keep in focus
  - pixel outside of focus range? blur
    - Blur, way 1 = average with neighboring pixels kernel size ~= amount of blur
    - Blur, way 2 = compute MIP-map of RGB image, use lower MIP-map level with bilinear interpolation

106

HDR - High Dynamic Range (limited Dynamic Range)

107

HDR - High Dynamic Range in a nutshell

- Screen space technique:
- First pass: fill the off-screen buffer like a normal rendering, EXCEPT use lighting / materials value that are HDR
  - so, RGB of final pixel values not in [0..1]
  - e.g., sun *emits* light with RGB [15.0 , 15.0 , 15.0]:

>1 = "overexposed"!
i.e., "whiter than white"
(here: 15 times brighter than the maximal screen brightness)

- Second pass:
  - Make values >1 *bleed* over neighboring pixels
  - i.e.: overexposed pixels lighten neighbors pixels
  - Result: halo effect

108

NPR rendering: e.g.: simulated pixel art

img by Howard Day (2015)

118