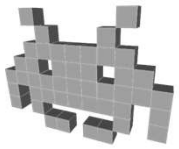
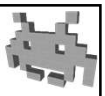
3D Videogames
Università degli Studi di Milano



Rendering in 3D games (bridge lecture)

1

Course Plan



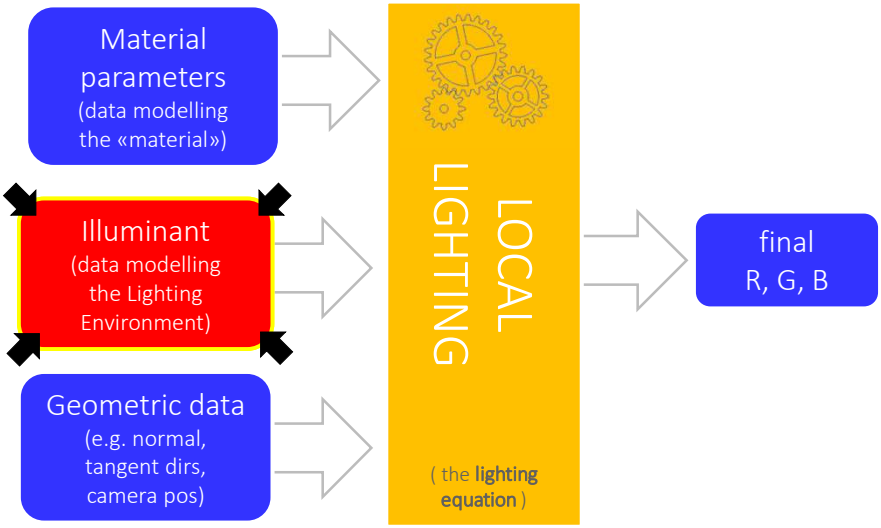
lec. 1: **Introduction**
lec. 2: **Mathematics** for 3D Games
lec. 3: **Scene Graph**
lec. 4: Game **3D Physics** +
lec. 5: Game **Particle Systems**
lec. 6: Game **3D Models**
lec. 7: Game **Materials**
lec. 8: Game **Textures**
lec. 9: Game **3D Animations**
lec. 10: **3D Audio** for 3D Games
lec.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

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»

★ bridge lectures

2

Part I: modelling the Light environment



3

Approaches to model the light environment in 3D games

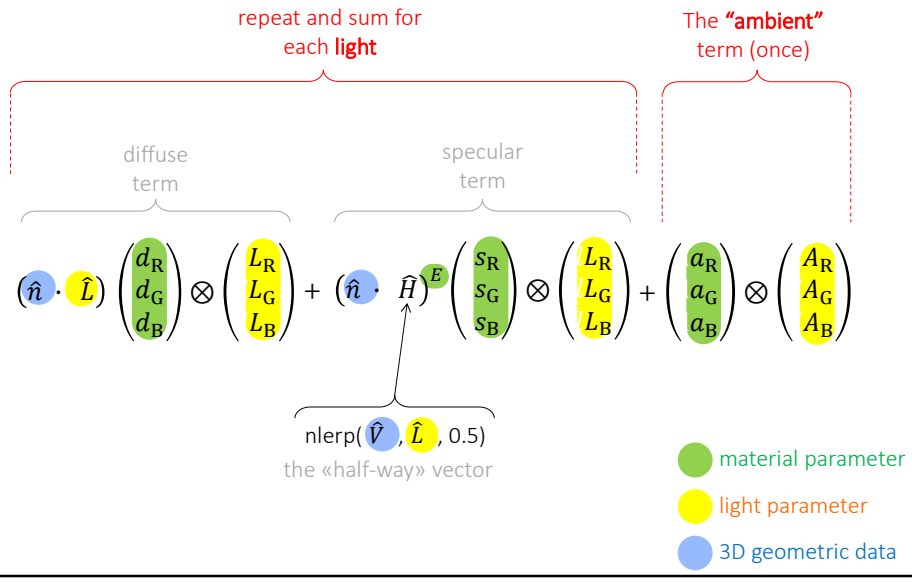
We are about to discuss three different approaches:

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

(They can be, and are, used jointly)

4

A simple lighting equation (recap)



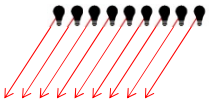
5

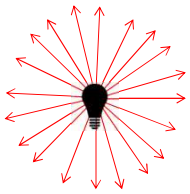
Discrete illumination environments: a set of individual *light sources*

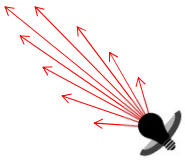
- a finite set of "light sources"…
  - not too many (e.g. ≤16)
  - if more, can be assigned "priorities" to pick a subset for every object being rendered
- each light sits in a node of the scene-graph!
- each light can be of one type…

6

Lights sources types & their attributes







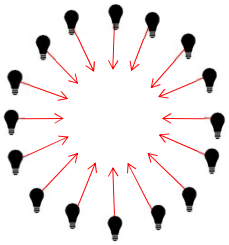
directional lights
- intensity / color
- **direction**

point lights
- intensity / color
- **position**
- falloff function (opt.)

spot-lights
- intensity / color
- **position**
- **direction**
- falloff function (opt.)
- angular falloff funct.
- "cookie" texture

7

Lights sources types & their attributes

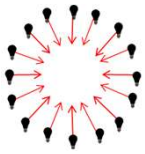


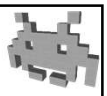
ambient light
- intensity / color
- *(that's it)*

but can be shadowed AMBIENT OCCULSION term.
E.g., stpred per-texel or per-vertex

8

Ambient light

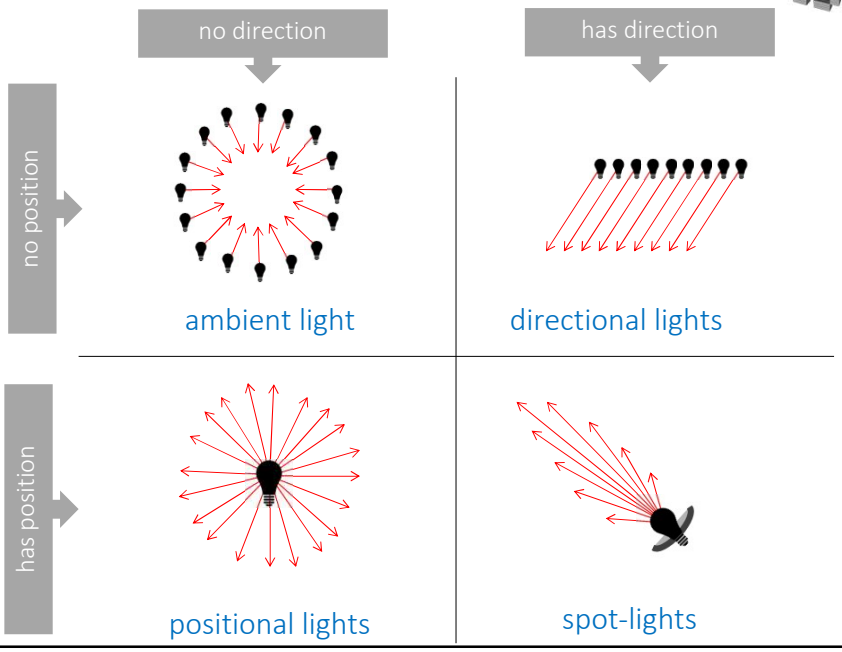




- models other all minor light sources + bounces
  - light incoming “from every direction at every position”
- examples:
  - in an overcast outdoor scene: it’s *high*
    - dim shadows, flat looking lighting:
      every photographs’ favorite for portraits!
  - in realistic outer space: it’s *zero*
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

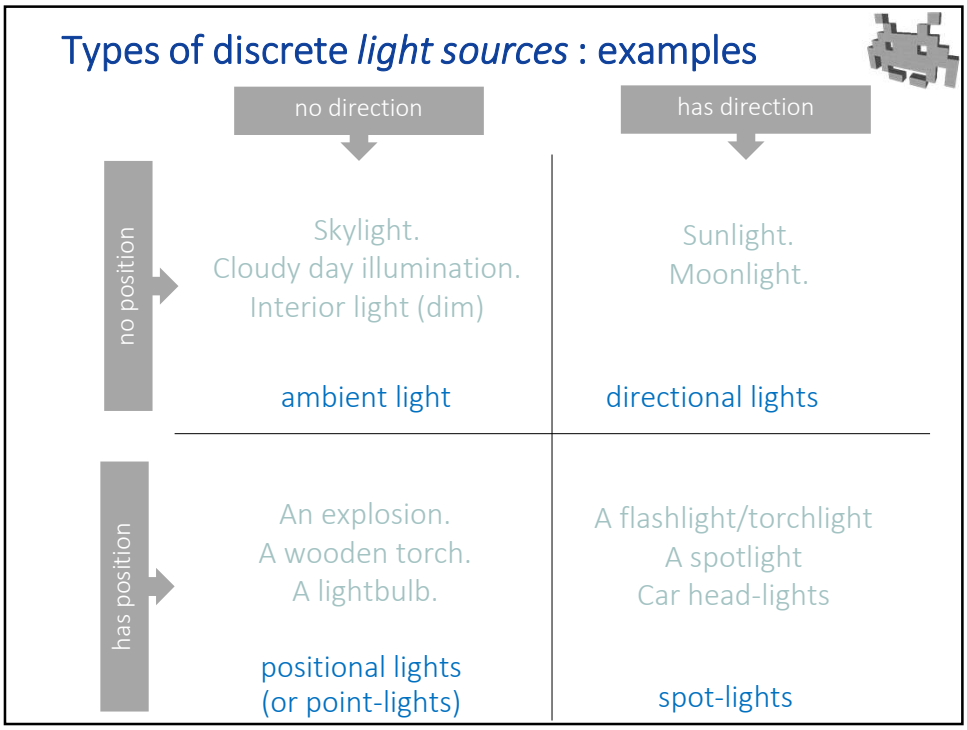
9

Types of discrete *light sources*



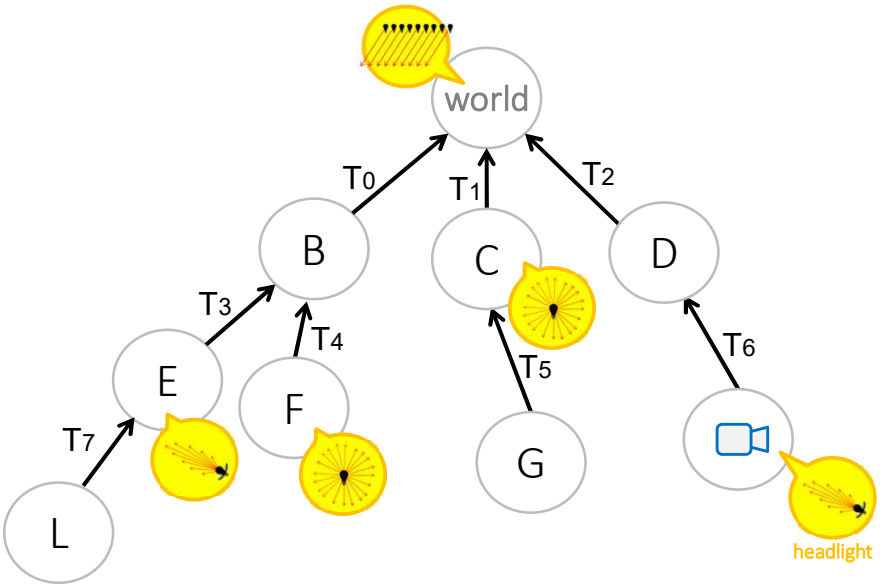
10

Types of discrete *light sources* : examples



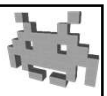
11

Discrete illumination environments



12

Distance fall-off functions
for positional lights & spotlights

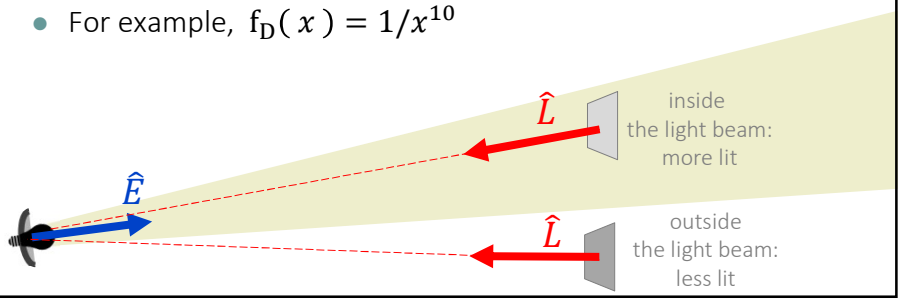


- The light intensity of positional lights and spotlights can be dimmed down with distance from light-pos $\mathrm{P_L}$ to the pos of the fragment being lit $\mathrm{P_P}$, scaling it by some positional «fall-off» function
$$\mathrm{f_P}(\|\mathrm{P_L} - \mathrm{P_P}\|)$$
  - In the real physical world,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

13

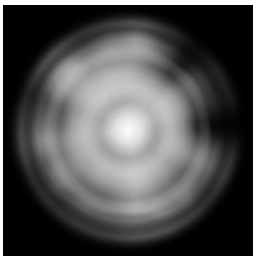
Angular fall-off functions
for spotlights

- 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
$$\mathrm{f_D}(-\hat{E} \cdot \hat{L})$$
  - For example, $\mathrm{f_D}(x) = 1/x^{10}$



14

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



as an alternative to angular fall-off functions

15

In the lighting equation…

repeat and sum for each **discrete light source**

the one **"ambient" light**

diffuse term

specular term

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

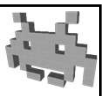
Light incoming direction
For directional lights: just a constant.
For a point- or spot-light: $\hat{L} = \frac{P_L - P_P}{\|P_L - P_P\|}$
pos of light, pos of lit point

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

parameter = of the light

16

Types of discrete lights (a summary)

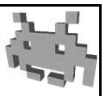





	Ambient light	Directional light	Positional light	Spotlight
geometry	*(nothing)* *(assumed at infinite)*	Direction (versor) *(assumed at infinite)*	Position (point)	Position (point) & Direction (versor)
can be dimmed by	-	-	Falloff function	Falloff function Angular falloff function "Cookie" texture
can 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?			

17

In which space to compute the lighting?



repeat for each light source

diffuse term + specular term + ambient term + emission term

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

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

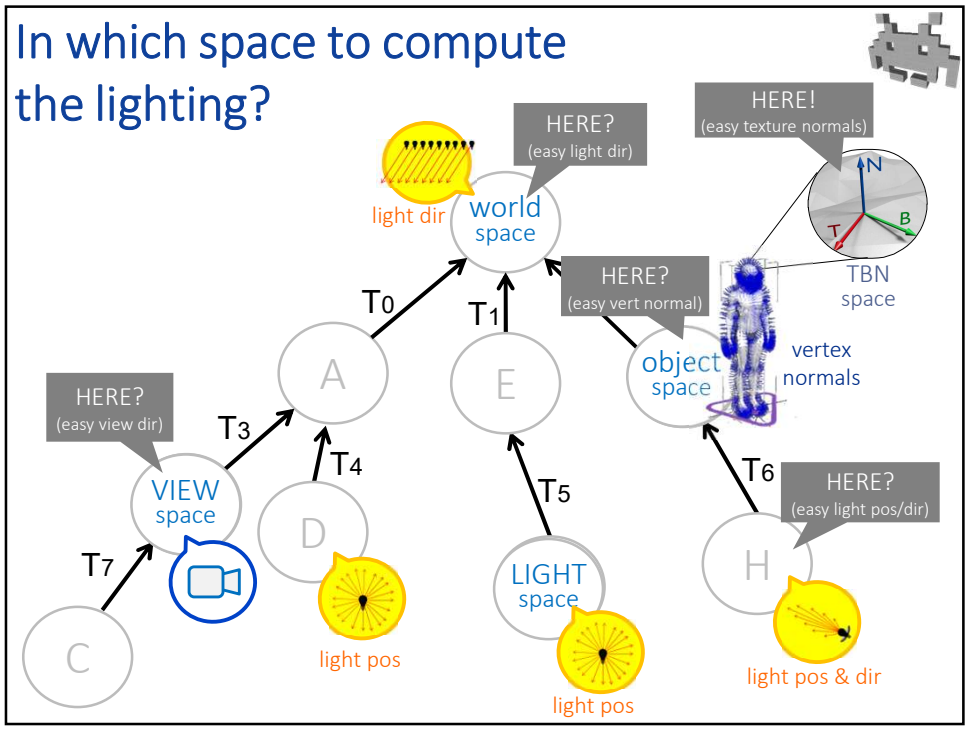
nlerp($\hat{V}$, $\hat{L}$, 0.5)

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*

18



19

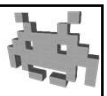
In which space to compute the lighting?

- 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…
- 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

for anisotropic materials

20

Discrete illumination environments Summary



- Pros:
  - simple to re-position / reorient individual light sources
    - both at design phase, or dynamically (at game exec)
  - good model of illuminants as:
    - explosions (positional lights)
    - car lights (spot-lights lights)
    - sun direction (directional light)

    main illuminants of the scene!
  - relatively easy to compute (hard, soft) shadows for them

    see shadow map later
- Cons:
  - each light source requires extra processing … for each pixel!
    - therefore: hard limit on their number. Prioritize them
    - 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

21

Continuous lighting environments

- In general terms, a lighting environment is a function:

  set of unit directions

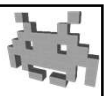
$$f : \Omega \longrightarrow \mathbb{R}$$

  or $f : \Omega \longrightarrow \mathbb{R}^3$ if we want colored lights

- $f(\hat{d})$ ≡ intensity/color of light coming form dir $\hat{d} \in \Omega$
- With discrete light environments:
  - we define f only over a few $\hat{d}$, with individual lights
  - f is a constant for every other $\hat{d}$, with ambient light
  - note: f is potentially defined differently in every point in the scene (when positional light sources are used). Cool!
  - note: f is dynamic: it's easy to move & turn on/off & change lights from a frame to the next. Cool!
  - But its not super realistic lighting env.

22

Continuous lighting environments



- In general terms, a lighting environment is a function:

set of unit directions

$$f : \hat{\Omega} \longrightarrow \mathbb{R}$$

or $f : \Omega \longrightarrow \mathbb{R}^3$ if we want colored lights

- $f(\hat{d}) \equiv$ intensity/color of light coming form dir $\hat{d} \in \Omega$
- Let's see two ways to encode a continuous f instead
  - Way 1: environment maps
    (just sample it, store it as a texture)
  - Way 2: with basis functions
    (good for very smooth lighting envs)
  - note: for now, we assume f is the same in everywhere in the scene
  - note: not easy to change f over time (there are ways – see later)
  - typically, a more realistic lighting env.

23

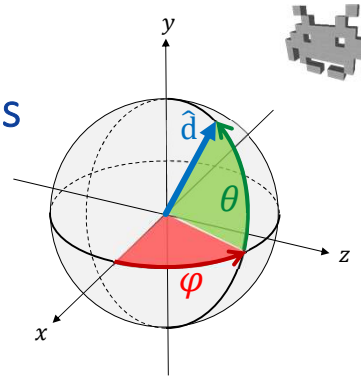
Densely sampled illumination environments

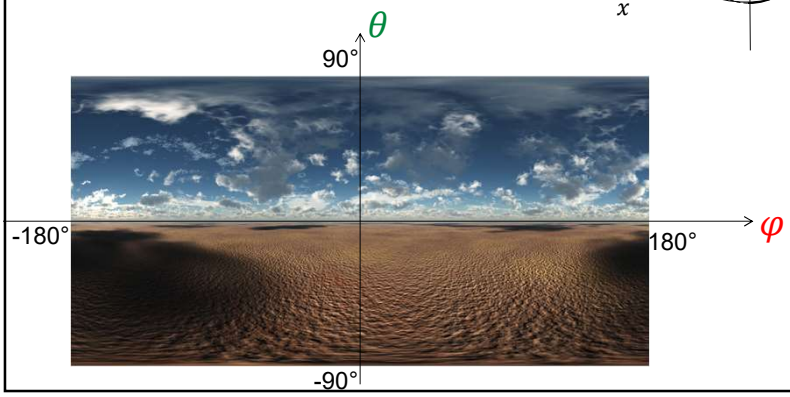
- A light intensity / color from each direction $\hat{d}$
- Asset to store that:
  "environment map" texture

24

Densely sampled illumination environments

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





25

Densely sampled illumination environments

- Aka "sky-map" texture
  - because it doubles as a texture for the "sky boxes"

26

Densely sampled illumination environments

- Environment map: (asset)
  a texture with a texel $\mathbf{t}$ for each direction $\hat{\mathbf{d}}$ (unit vector)
  - $\mathbf{t}$ stores the intensity/color of the light coming from direction $\hat{\mathbf{d}}$
- Q: how to determine u, v position of $\mathbf{t}$ for a given $\hat{\mathbf{d}}$?
  - 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!)

27

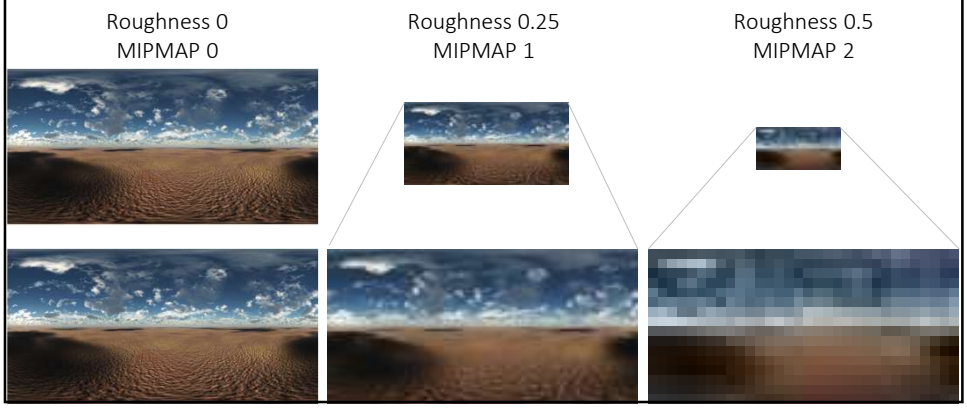
Environment map (asset)

- A texture with a texel $\mathbf{t}$ for each direction $\hat{\mathbf{d}}$
  - $\mathbf{t}$ stores the light coming from direction $\hat{\mathbf{d}}$
  - 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

28

Environment map (asset): uses

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



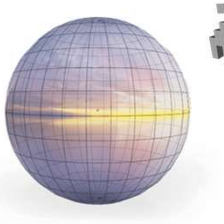
29

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

30

Skydome / skybox mesh



- A mesh encapsulating the entire scene
  - shows far away objects
  - note: for technical reason, the rendering doesn't show really far away objects – see "far plane clipping"
  - technically, it shows objects at ∞ distance
- Shaped as a large cube/sphere surrounding the scene
- Centered in the same node as the camera
  - so, it "follows the camera": "players can never reach it"
- But rotated as the world frame
  - not as camera!
  - (similar to: the transform of the microphone node)
- The environment map doubles as a color texture over this mesh
  - The sphere is painted with the background objects (stars, sky, mountains on the horizon, etc)

32

Light environments: using Basis Functions

- Lighting environment:
  a *continuous* function $f : \Omega \rightarrow \mathbb{R}$

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

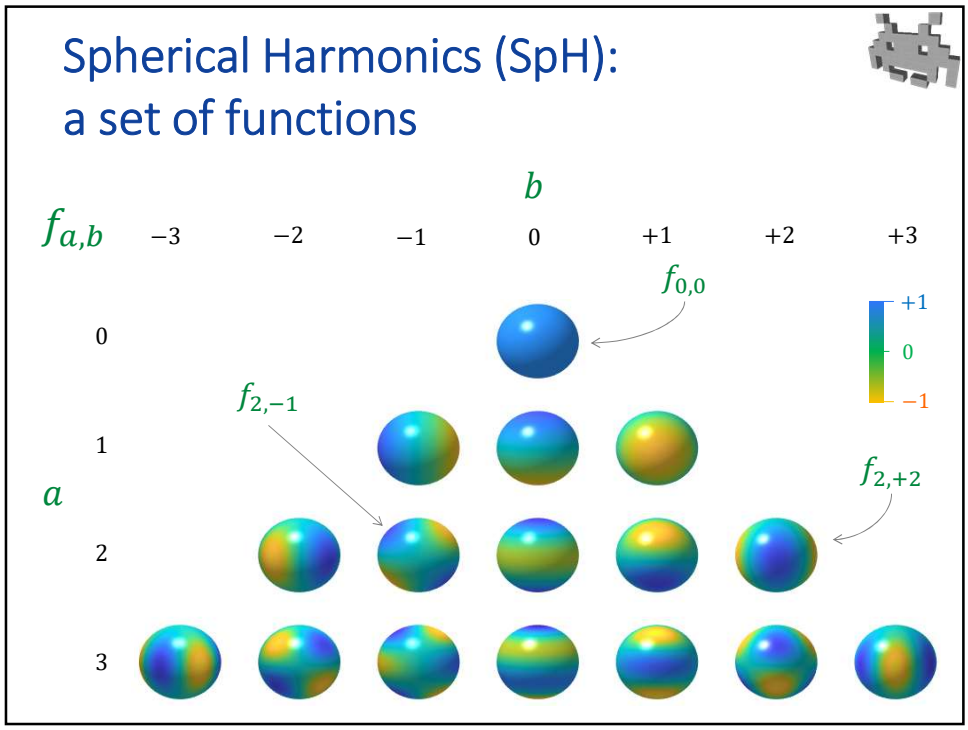
  or R^3 if RGB colored light
- $f(\hat{v})$ = amount of (rgb) light coming from direction $\hat{v}$
- Store f through basis functions

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

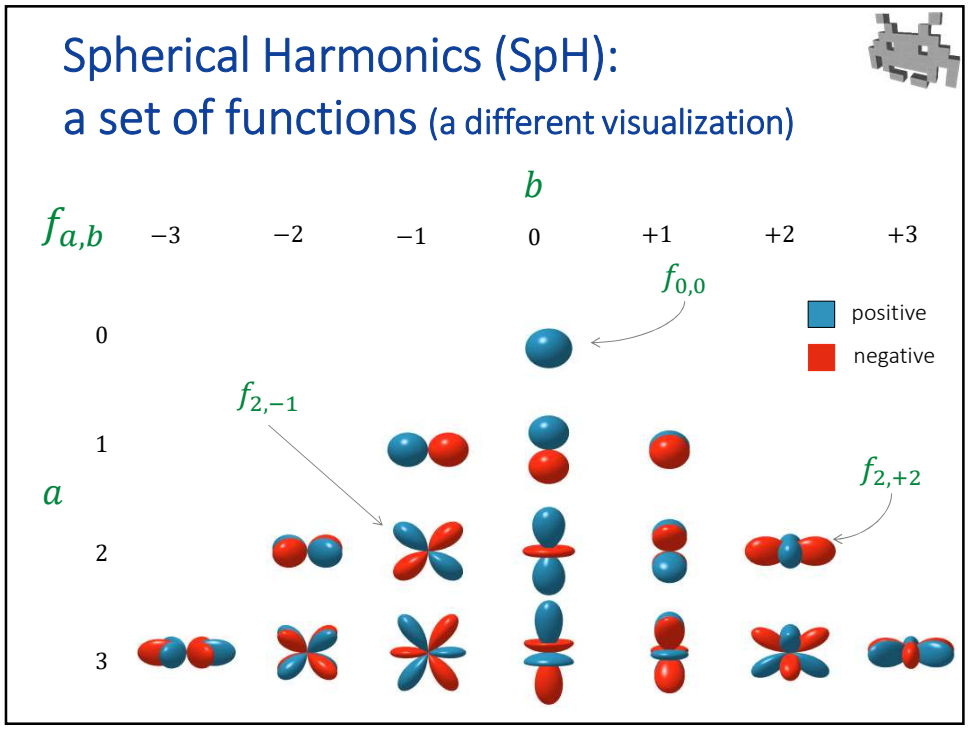
$$f(\hat{v}) \cong a_{0,0} \cdot f_{0,0}(\hat{v}) + a_{1,-1} \cdot f_{1,-1}(\hat{v}) + a_{1,0} \cdot f_{1,0}(\hat{v}) + a_{1,+1} \cdot f_{1,+1}(\hat{v}) + \cdots$$

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

34


Spherical Harmonics (SpH):
a set of functions
$f_{a,b}$
b
−3
−2
−1
0
+1
+2
+3
a
0
1
2
3
$f_{0,0}$
$f_{2,-1}$
$f_{2,+2}$
+1
0
−1

36


Spherical Harmonics (SpH):
a set of functions (a different visualization)
$f_{a,b}$
b
−3
−2
−1
0
+1
+2
+3
a
0
1
2
3
$f_{0,0}$
$f_{2,-1}$
$f_{2,+2}$
positive
negative

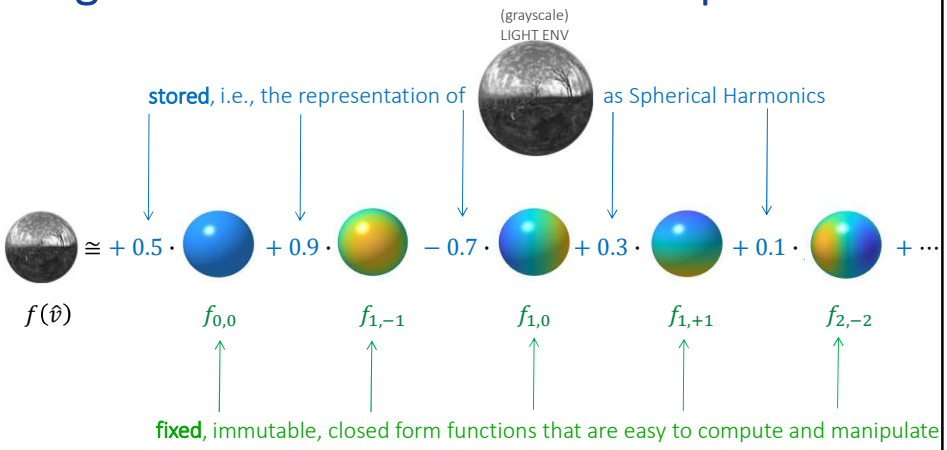
37

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 $a \geq 0$, $-a \leq b \leq +a$
  - $f_{0,0}(\hat{v}) = 1$ 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 evaluate, integrate, etc

38

Light probes: Light environment stored with SpH

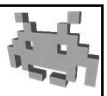


f is stored as $(+0.5, +0.9, -0.7, +0.3, 0.1, \ldots)$

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

39

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

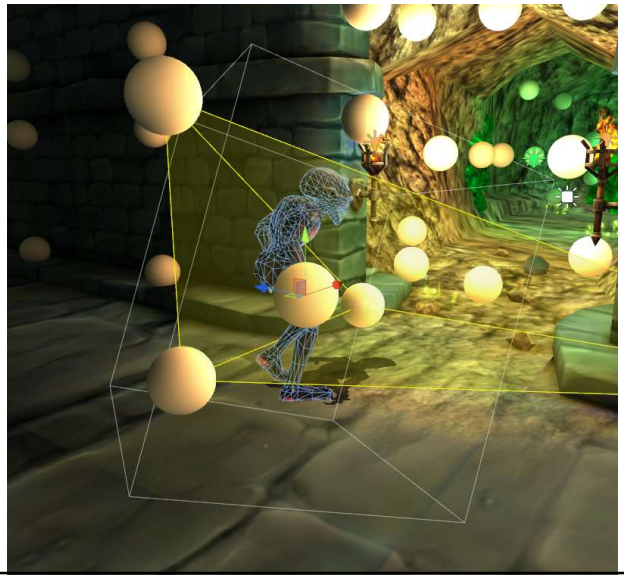
40

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

41

Light probes (position-dependent lighting env)

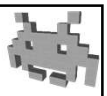


42

Light probes (position-dependent lighting env)

43

Part II: basics of GPU-based rendering



- Summary:
  - a brief summary of rasterization-based rendering (currently, almost universally adopted in 3D games) (somehow threatened by novel “ray tracing” based approaches)
  - the graphic pipeline and its programmable parts
  - depth-maps
  - double buffering
  - deferred shading
  - multi-pass techniques in general (with exammples)

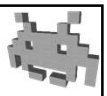
46

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

47

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

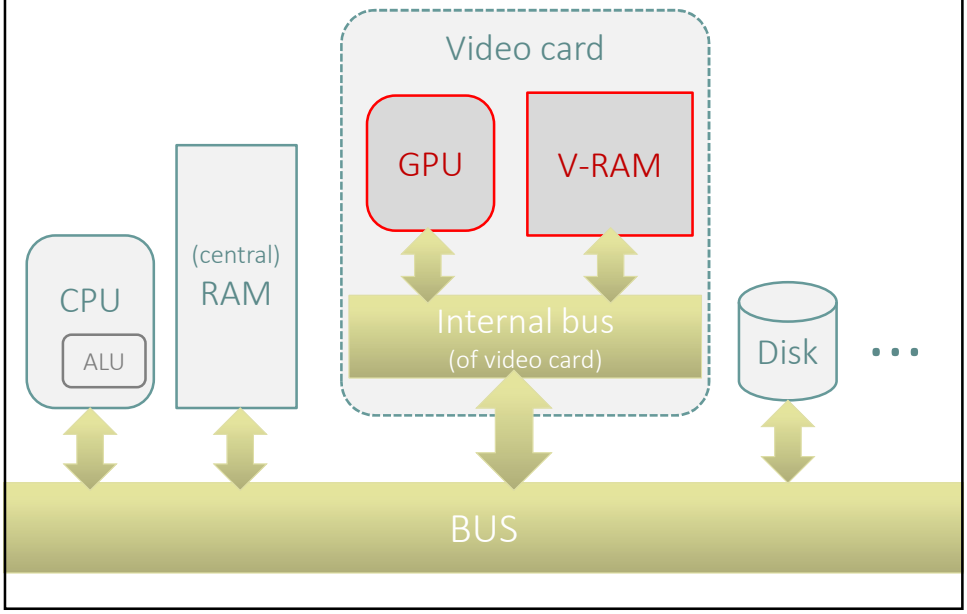
THE MESH ASSET

THE MATERIAL ASSET

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

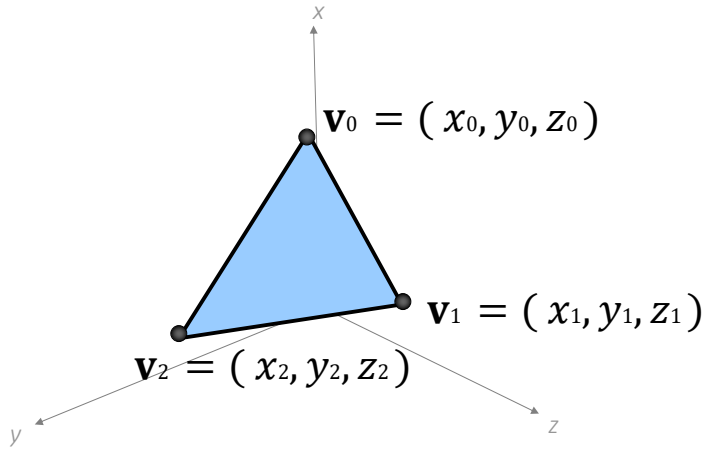
48

Graphic card



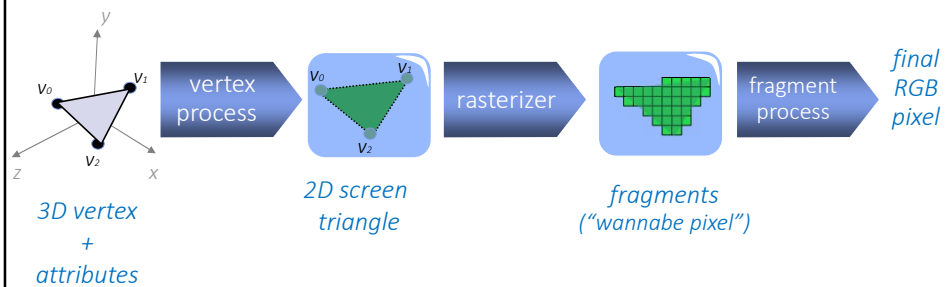
50

Rendering of a mesh = rasterization of all its triangles



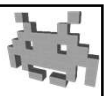
51

GPU pipeline – a simplified conceptual version



52

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

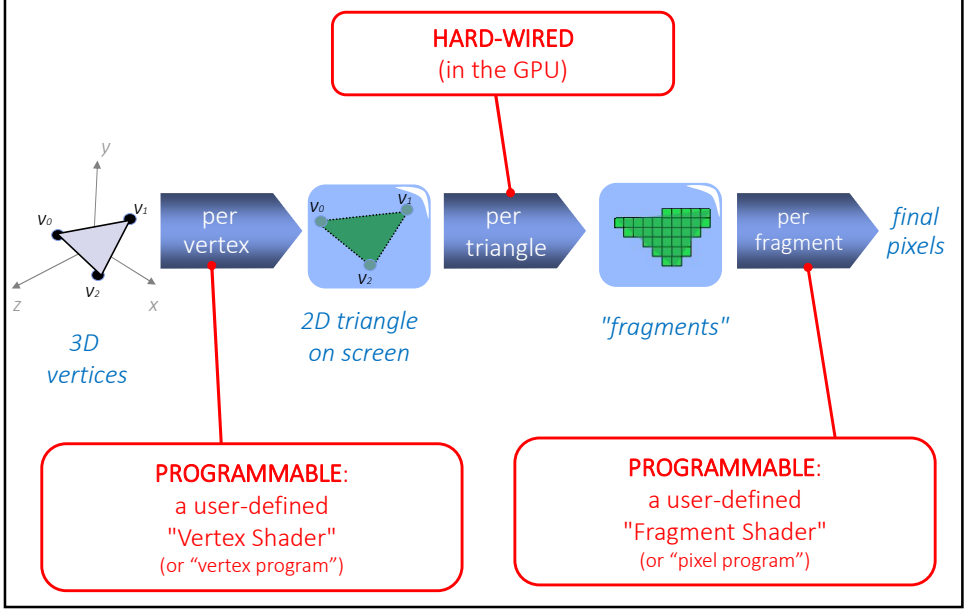
57

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

58

Rasterization-Based Rendering



60

Rasterization based rendering: typical tasks 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

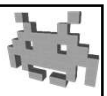
hard wired
- **Per triangle:** (rasterizer)
  - back-face culling discard of back-facing triangles
  - rasterization creation of fragments
  - interpolation of per-vertex attributes ←*nota bene!*

the Fragment Shader
- **Per fragment:**
  - texture mapping: 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

61

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 wait for to next stage to be ready for their output)
    - stages after the bottleneck are «starved» (they wait for the input from previous stage)
- Terminology about bottlenecks: (in CG)
  - "per vertex" stage is the bottleneck: the app is goemetry-limited («it cannot process geometry fast enough»)
  - "per fragment" stage is the bottleneck: the app is fill-limited («it cannot fill the screen buffer with pixels fast enough»)

MORE COMMON CASE, FOR GAMES

- Performaces (rendering FPS) of a game only impoves if computational load is lifted from the bottleneck phase
  Examples:
  - using all meshes at LOD 2 instead of LOD 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

62

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

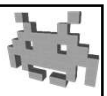
THE MESH ASSET

THE MATERIAL ASSET

- issue the Draw-call

63

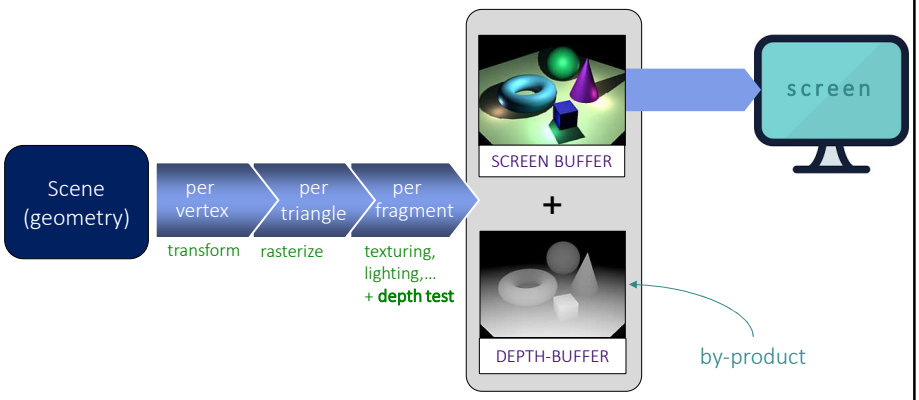
Programming languages for writing shaders



- 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)

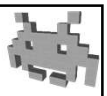
64

basics: Depth buffer



65

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 sent 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 occluded by that something)
  - see the 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)

a 2D array of RGB values of some resolution

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

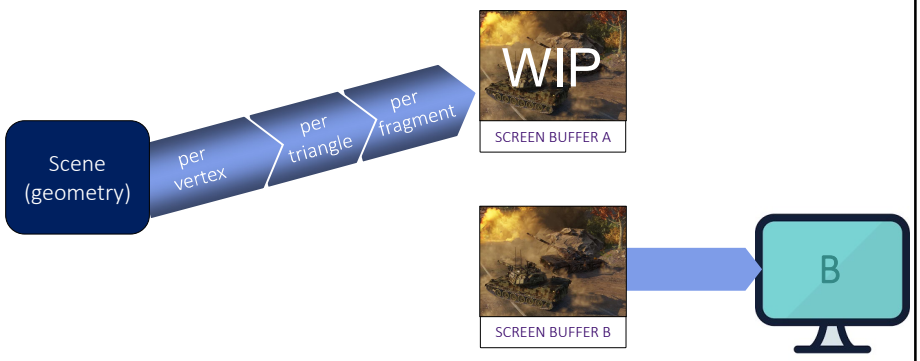
66

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: no need to double the depth-buffer

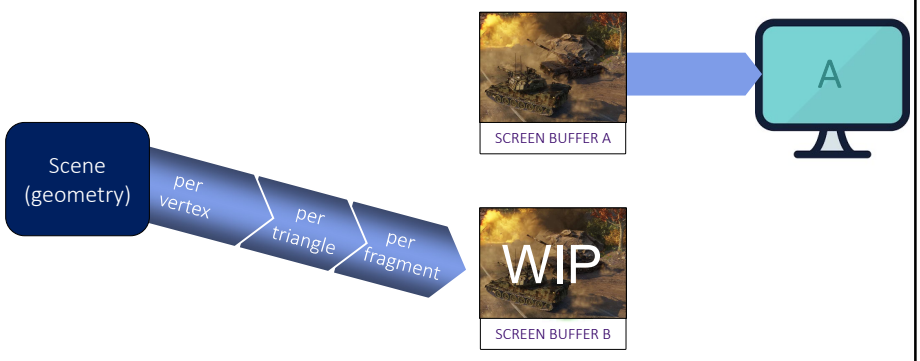
67

basics: Double Buffering



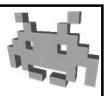
68

basics: Double Buffering



69

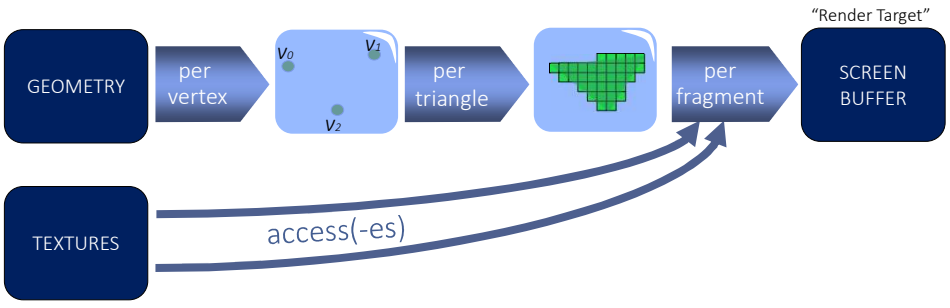
Off-screen buffers



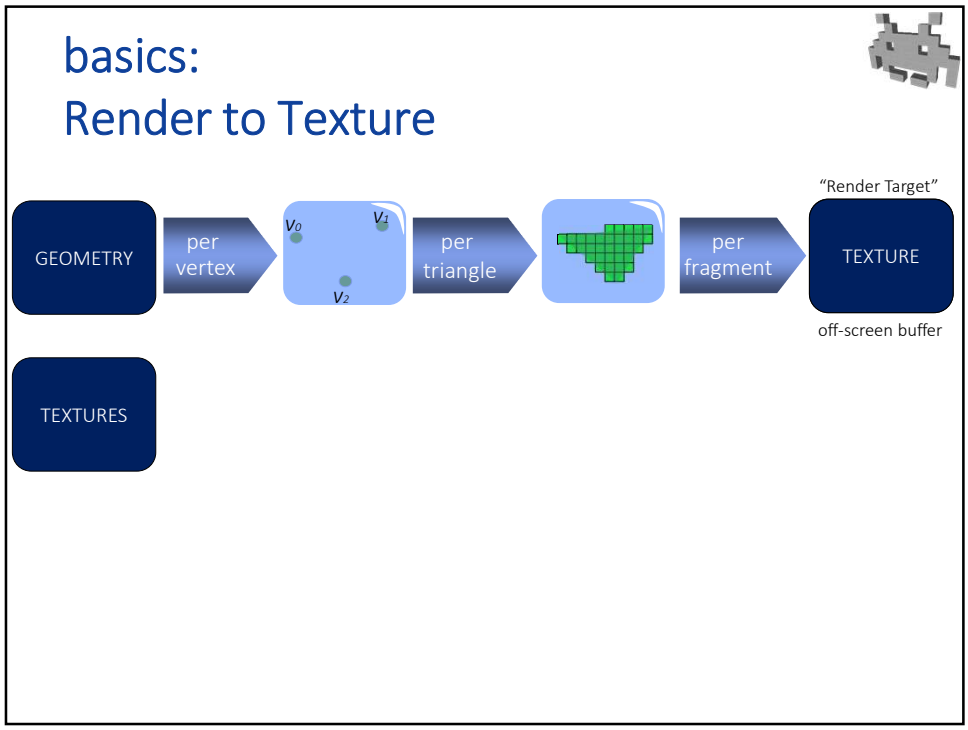
- The rendering produces a screen buffer (2D array of RGB pixel) that is sent to the screen and is made visible to the player
- Any buffers that is used internally but and not sent to the screen is called an off-screen buffer
  - Example: the depth buffer (2D array of depth values)
  - Example: the back-screen buffer (double buffering techniques)
- Many rendering techniques are based on producing an off-screen buffer *then* using it in another pass
  - For example, to re-using it a texture in another rendering pass

70

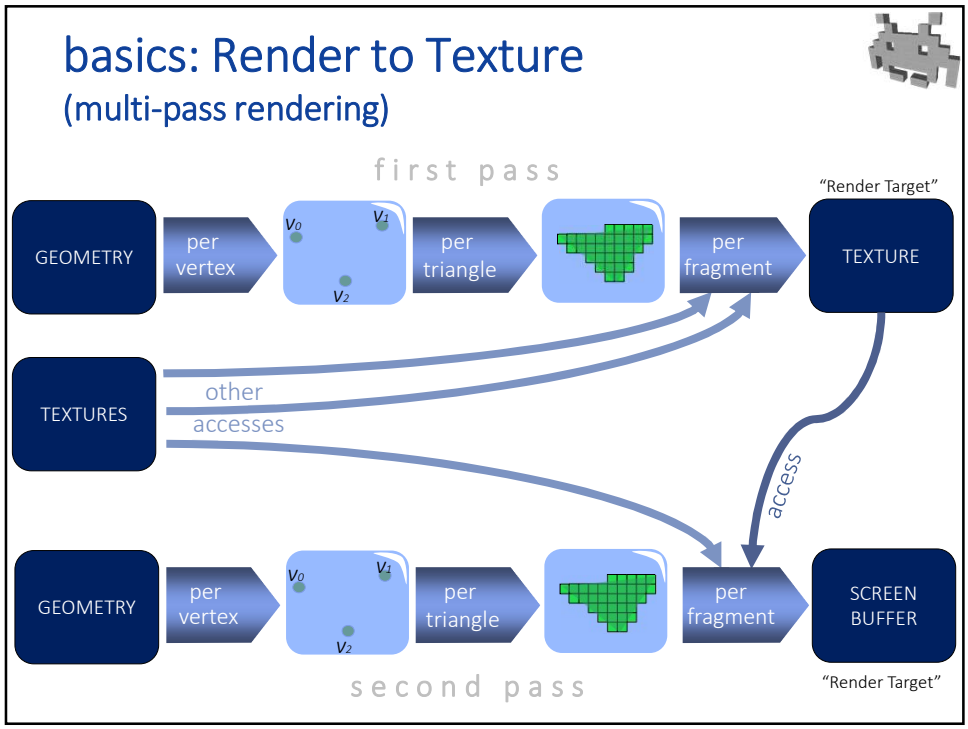
Texture access: it's in the per fragment process



72


basics:
Render to Texture
GEOMETRY
per vertex
per triangle
per fragment
"Render Target"
TEXTURE
off-screen buffer
TEXTURES

73


basics: Render to Texture
(multi-pass rendering)
first pass
GEOMETRY
per vertex
per triangle
per fragment
"Render Target"
TEXTURE
TEXTURES
other accesses
access
GEOMETRY
per vertex
per triangle
per fragment
SCREEN BUFFER
second pass
"Render Target"

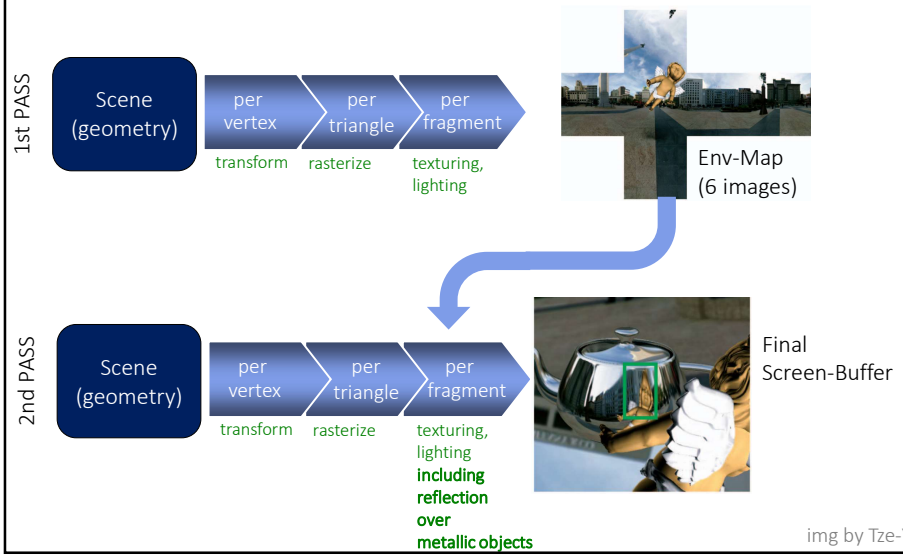
74

Multipass rendering techniques
(a wide class of rendering techniques)

- 1st pass: fill an internal 2D buffer
  - an "off-screen" buffer
  - the output of this rendering, i.e. its "render target"
  - normally, the render target is the "screen buffer" (the buffer shown to the screen), but not in this case
- 2nd pass: fill the final screen buffer
  - using the just-computed off-screen buffer as a 2D texture
  - the 1st pass was… "rendering to 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!

75

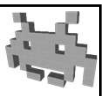
Example: metallic reflections of *dynamic* scenes



img by Tze-Yiu Ho

78

Main rendering algorithms: two classes of approaches



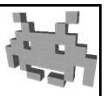
- Forward rendering
- Deferred shading

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

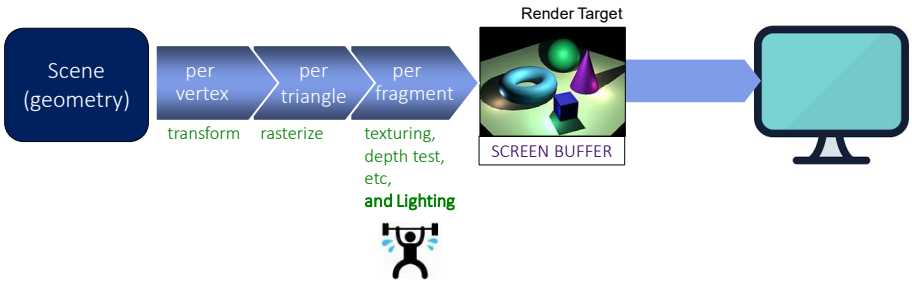
- Which approach to use?
  - Both are employed by games
  - Basilar choice! Implementation of all other rendering algorithms changes accordingly.

79

Main rendering algorithms: two classes of approaches



- Forward rendering

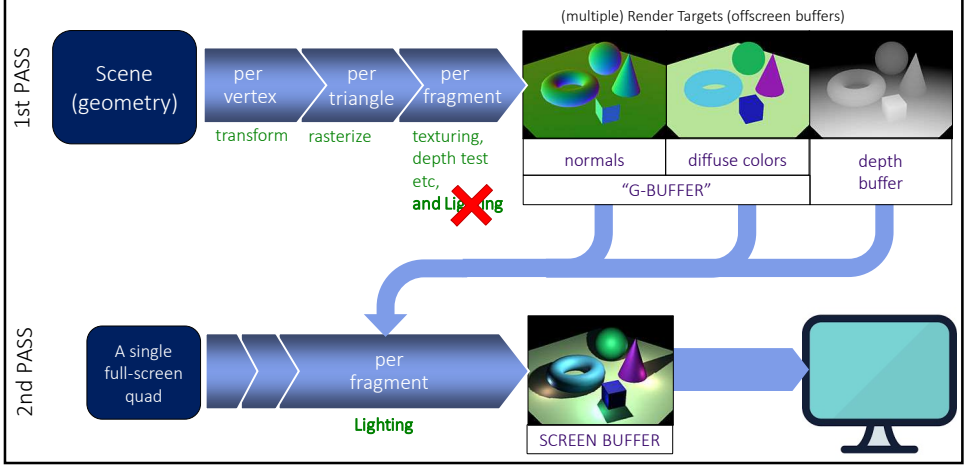


80

Main rendering algorithms: two classes of approaches

- Deferred shading

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



81

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

82