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: **Networking** for 3D Games ◗
lec. 11: **Artificial Intelligence** for 3D Games ●
lec. 12: **3D Audio** for 3D Games ●
lec. 13: **Rendering Techniques** for 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»

2

Example of a lighting equation

repeat and sum for each light source

add only once
(for the "ambient" light)

diffuse term

specular term

ambient term

emission term

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

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

the «half-way» vector
(avg of view dir and light dir)

material parameter

light parameter

geometry

4

Next: modelling the Light environment

Material parameters
(data modelling the «material»)

Illuminant
(data modelling the Lighting Environment)

Geometric data
(e.g. normal, tangent dirs, pos viewer)

LOCAL LIGHTING
(the lighting equation)

final
R, G, B

5

Approaches to model the light environment in 2D games

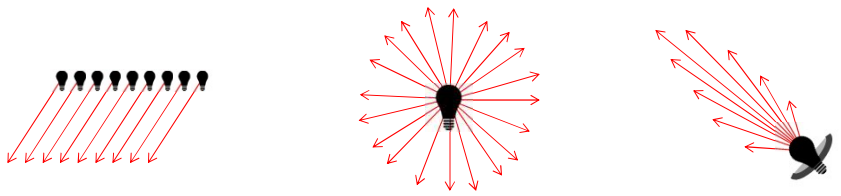
We are about to discuss three ways:

- 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 used jointly)

6

Discrete illumination environments. A set of light sources:



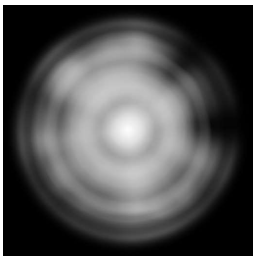
directional light
- intensity / color
- direction

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

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

7

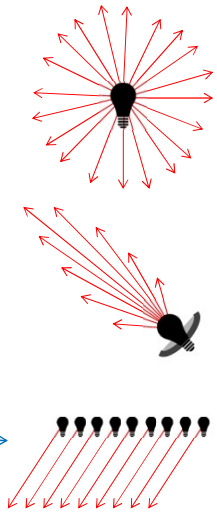
Cookie texture (for spot-lights)



8

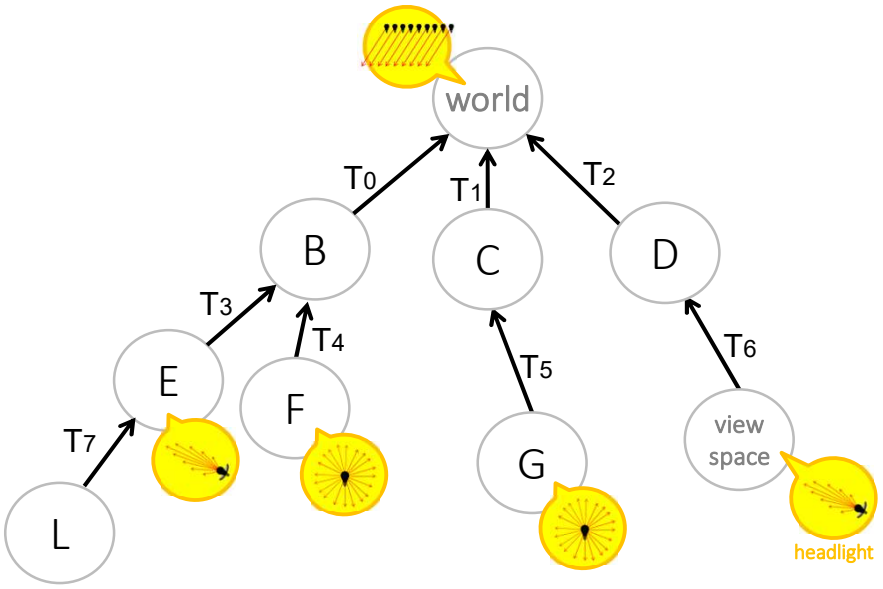
Illumination environments: discrete

- a finite set of "light sources"…
  - not many (e.g. ≤16)
- each one sitting in a node of the scene-graph
- each of a type:
  - point light sources
    - have: position
  - spot-lights
    - have: position, orientation, wideness (angle)
  - directional light sources
    - have: orientation only
- with attributes such as:
  - color / intensity
  - fall-off function (with distance)
  - max range, and more



9

Light sources in the scene graph



10

Example of a lighting equation

repeat coputation and sum
for each (positional, directional, spot-) light source

ambient light

diffuse term

specular term

ambient term

emission term

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

For a directional light: a constant.

For a point or spot-light: $\hat{L} = \frac{P_L - P_S}{\|P_L - P_S\|}$

pos of light

pos on surface

For a dir. light: consts. For point or spot-lights: attenuated by fall-off functions (of angle, dist)

material

light

geometry

11

Discrete illumination environments

- 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

main illuminants of the scene!

see shadow map later

- 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

12

Discrete illumination environments

a finite set of "light sources":

- A number of (directional | positional | spot-) lights
- Plus, one global "ambient light" factor
  - models other minor light sources + bounces
    - light incoming "from every direction at every position"
  - it's a distinct term in the lighting equation
  - 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)

13

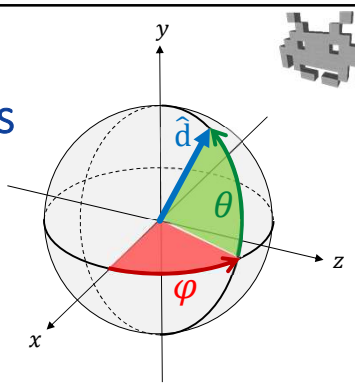
Densely sampled illumination environments

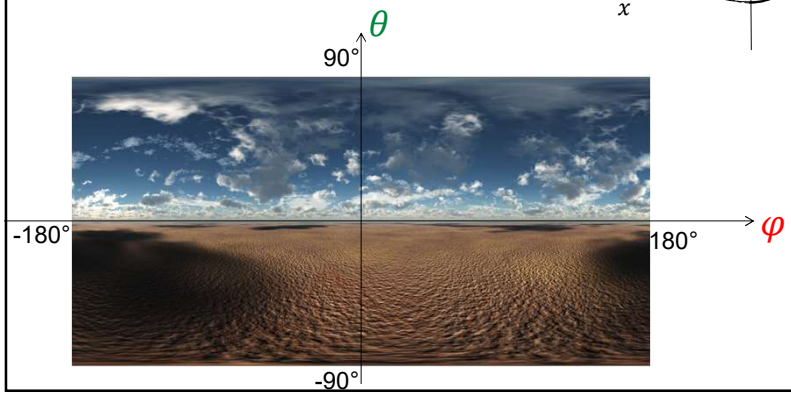
- A light intensity / color from each direction $\hat{d}$
- Asset to store that:
  "Environment map" texture

14

Densely sampled illumination environments

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





15

Densely sampled illumination environments

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

16

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 find 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!)

17

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

18

Environment map (asset): uses

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

Roughness 0
MIPMAP 0

Roughness 0.25
MIPMAP 2

Roughness 0.5
MIPMAP 4

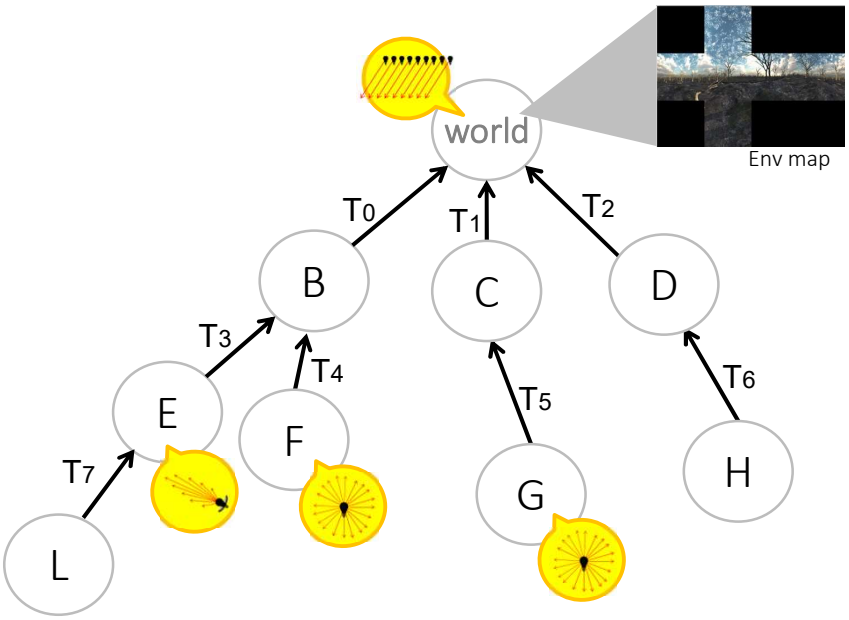
19

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 for the 3D sky-box / sky-dome

20

Lighting env in the scene graph



21

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

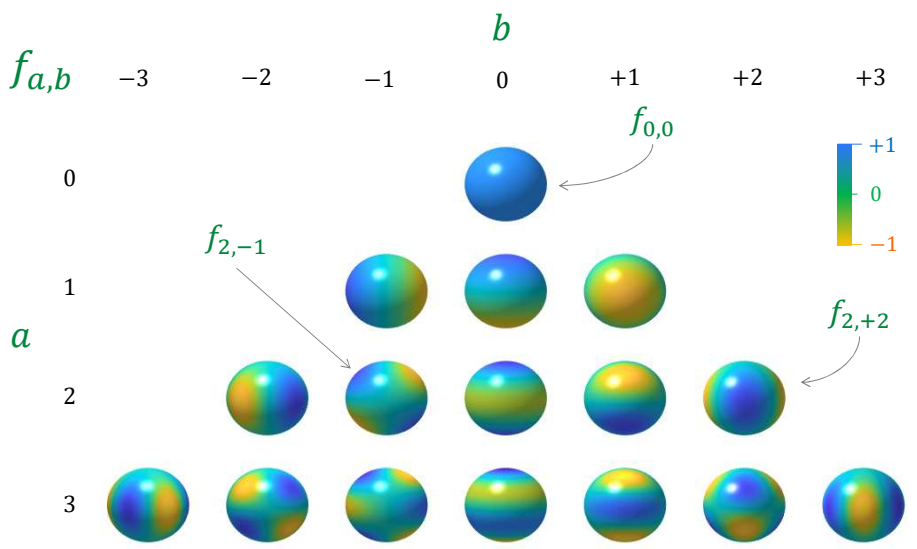
fixed spherical "basis" functions (always the same ones)

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

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

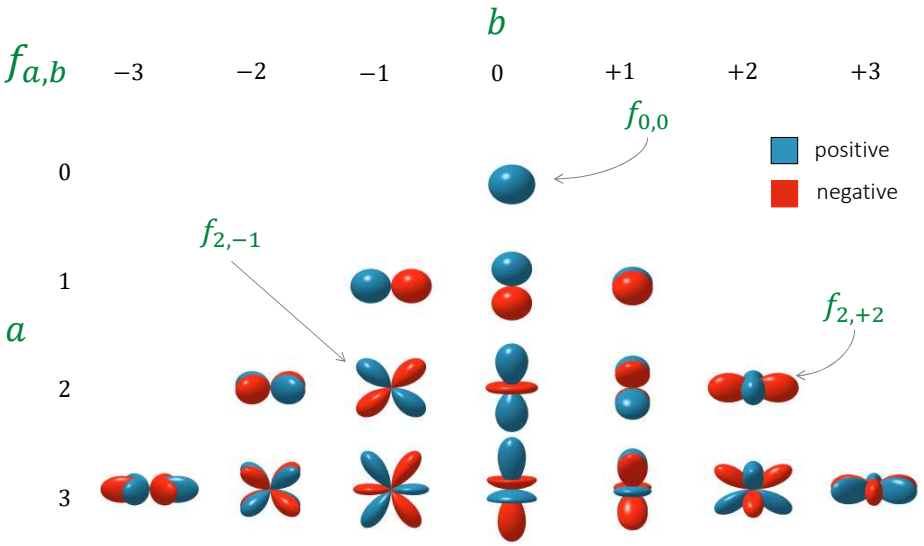
22

Spherical Harmonics (SpH): a set of functions



24

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



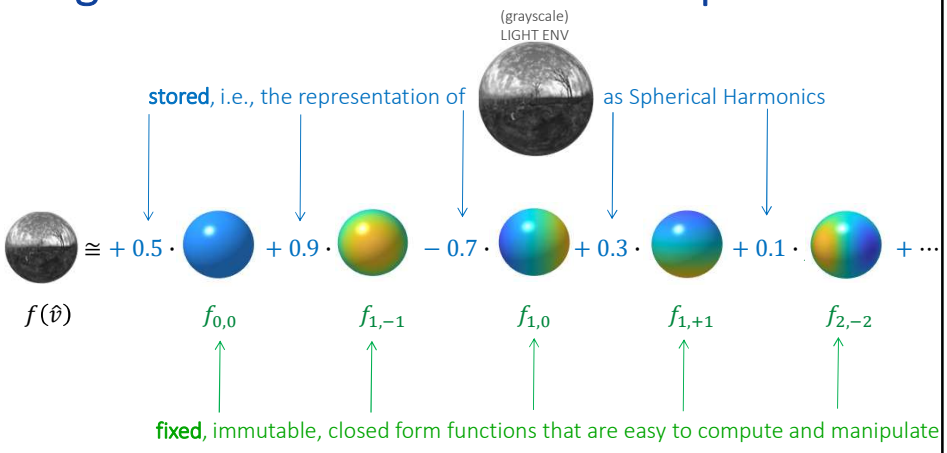
25

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 compute, e.g. integrate, etc

26

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)

27

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

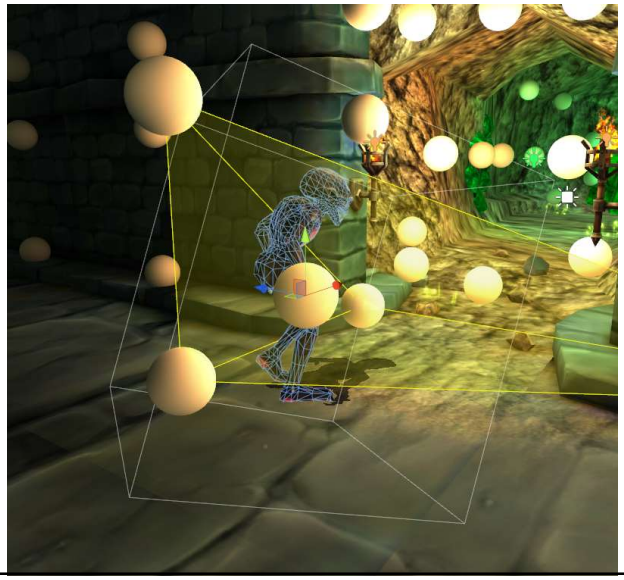
28

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

29

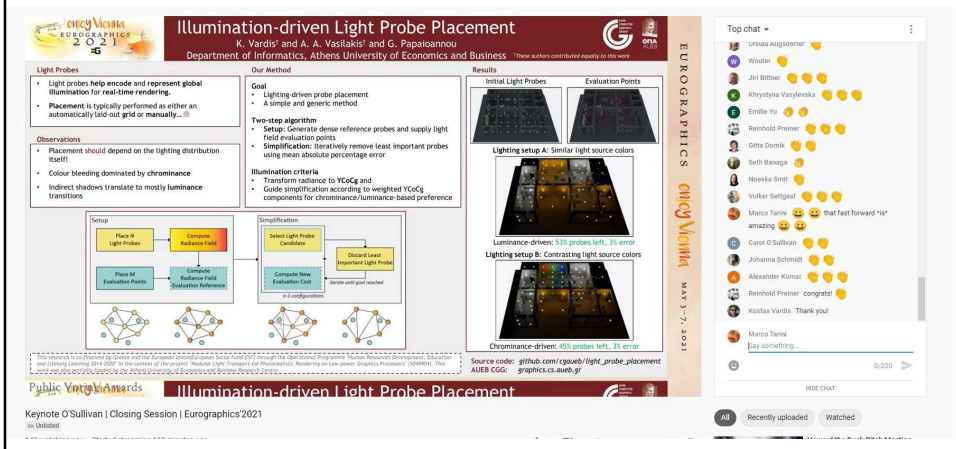
Light probes (position-dependent lighting env)



30

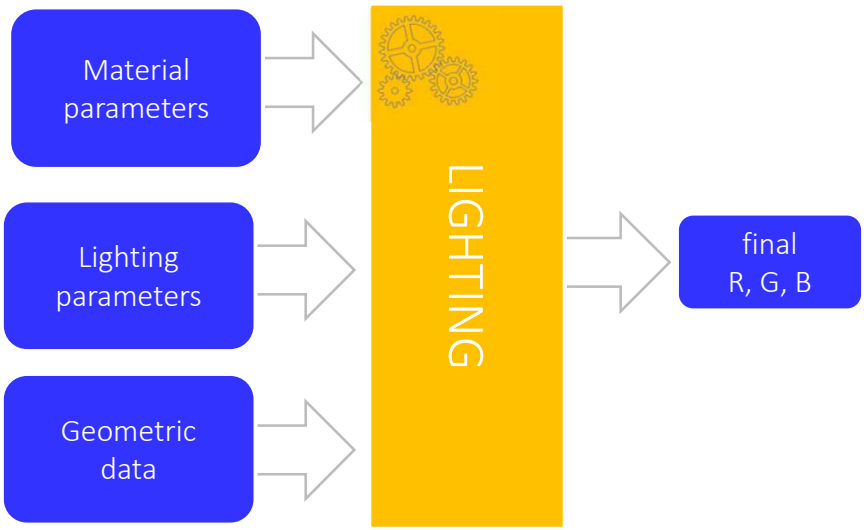
Light probes
(position-dependent lighting env)

31



32

How parameters are fed to the lighting equation



34

An example of a lighting model

repeat for each light source

diffuse term + specular term + ambient term + emission term

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

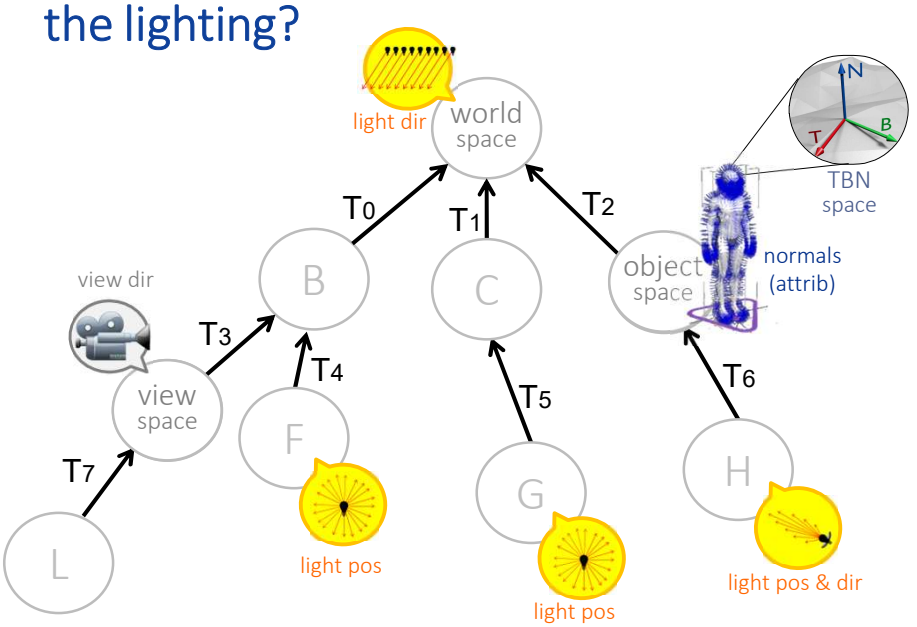
$\text{nlerp}(\hat{V},\hat{L},0.5)$

the «half-way» vector

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

36

In which space to compute the lighting?



37

In which space to compute the lighting?

- All versors that used in any operation 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

38

Rendering in 3D games

Part II: Common CG techniques in games

39

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

40

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

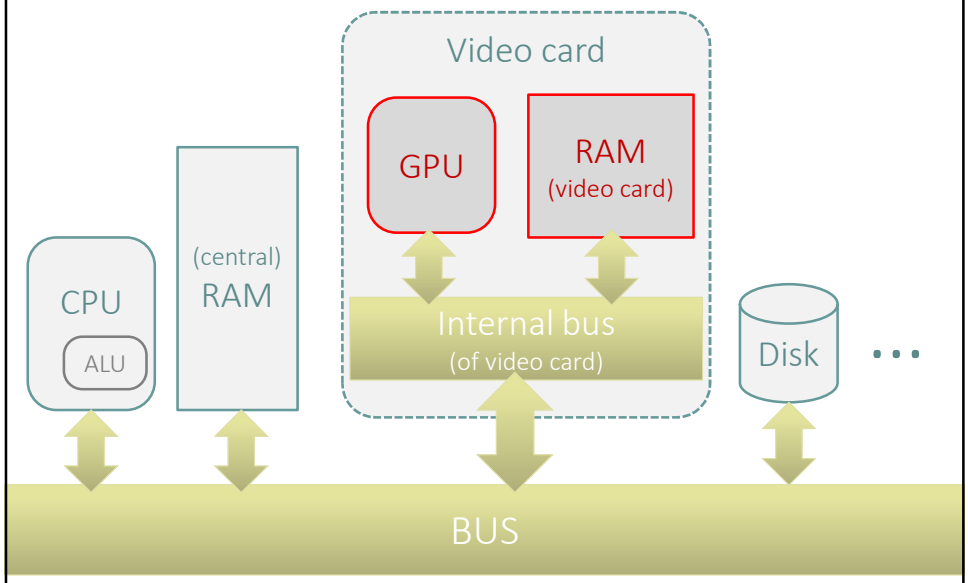
THE MESH ASSET

THE MATERIAL ASSET

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

41

Graphic card



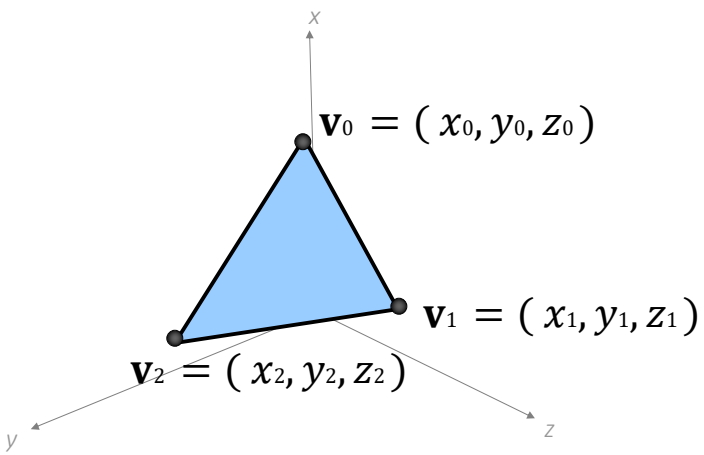
43

This lecture: a bird-eye view on…

- 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

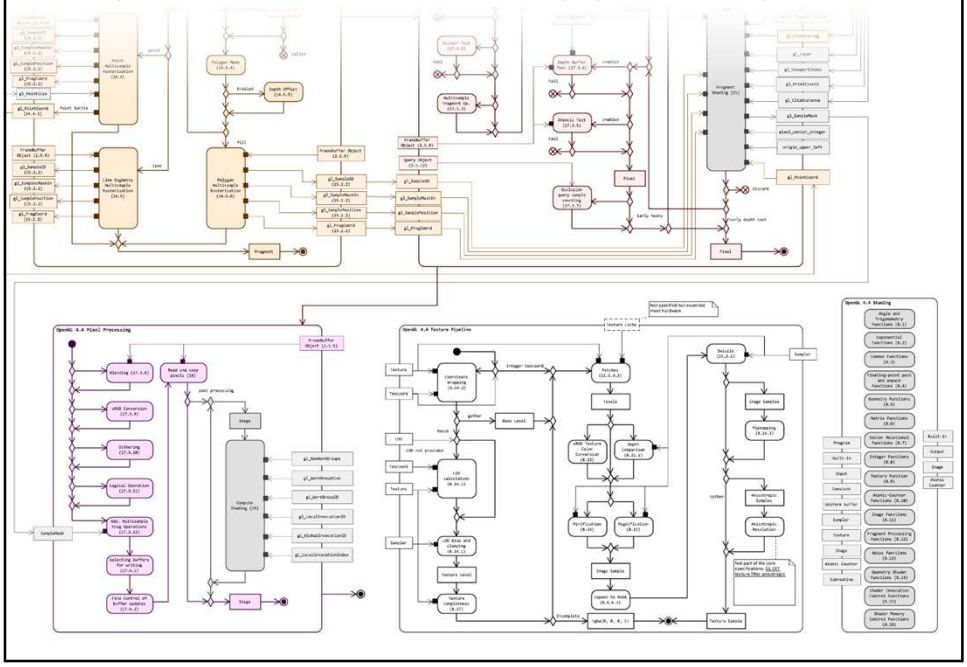
44

Rendering of a mesh = rasterization of all its triangles



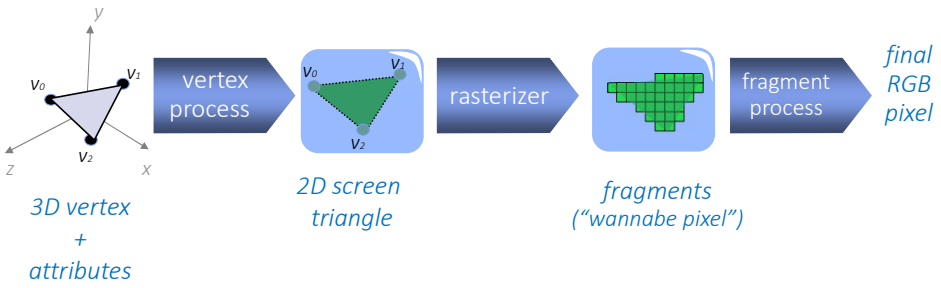
45

Example: full API for the GPU pipeline (OpenGL) 2/2



47

GPU pipeline – a simplified conceptual version



50

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

It's a SIMD architecture: Every step does the same processing on several input producing several output, all in parallel,

51

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

52

Rasterization based rendering: what is done in each step (examples)

the Vertex Shader
- Per vertex:
  - projection: transform from object space to screen space
  - skinning: transform from rest pose to current pose

hard wired
- Per triangle: (rasterizer)
  - rasterization
  - interpolation all per-vertex attributes ←*nota bene!*

the Fragment Shader
- Per fragment:
  - lighting: from normal + lights + material to RGB
  - texturing: i.e., textures are accessed in this stage
  - alpha-kill: (almost) fully transparent fragments are removed

hard wired
- Per fragment: (output combiner, after the fragment shader)
  - depth-test: occluded pixels are removed
  - alpha-blend: semi-transparent fragments are mixed with background

54

GPU pipeline – bottlenecks
(remarks and terminology)

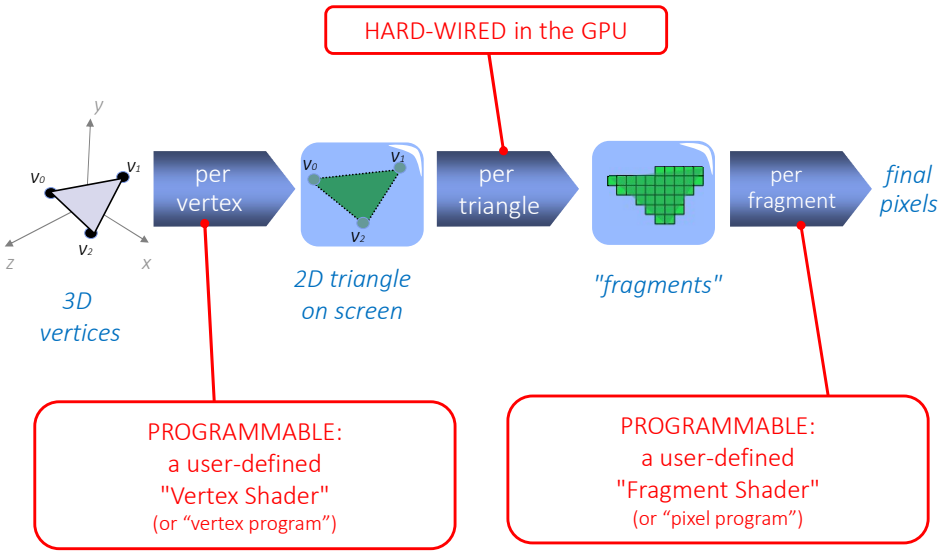
- Like in any pipeline, the process goes *as slow as its slowest stage*
  - i.e., the «bottleneck» of the pipeline determines the total speed
  - Any other stage is idle for part of the time (which is always a waste)
    - stages before the bottleneck are «chocked» (they cannot produce output because 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
  - Example: using all meshes at LOD 1 instead of one does not help a fill-limited app
  - Example: 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

55

Rasterization-Based Rendering



56

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

THE MATERIAL ASSET

57

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)

58

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

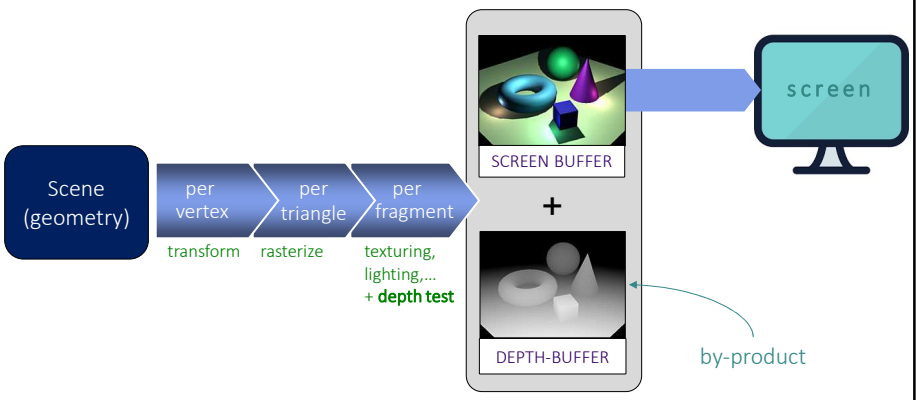
- Any rendering producing a screen-buffer …
  - which is sent to the screen
- …also produces a depth-buffer
  - as a by-product!
  - not set to the screen: it's an "offline" buffer
  - it's used during the rendering to determine occlusions and remove "hidden surfaces"
    (i.e. make what is behind something 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)

a 2D array of RGB values of some resolution

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

59

basics: Depth buffer



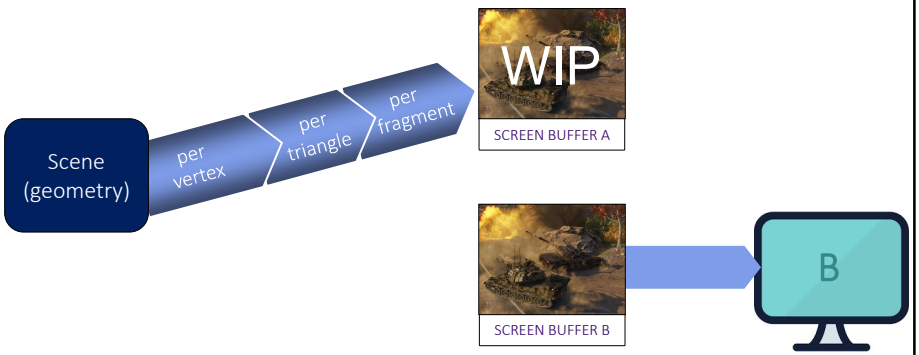
60

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 is not doubled

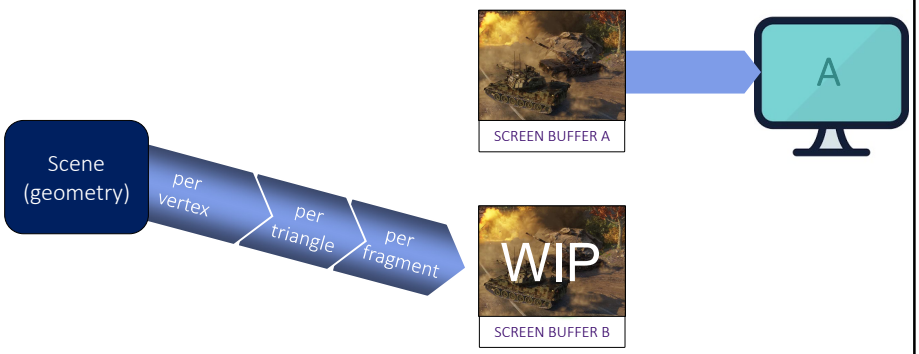
61

basics: Double Buffering



62

basics: Double Buffering



63

basics: Per-pixel lighting

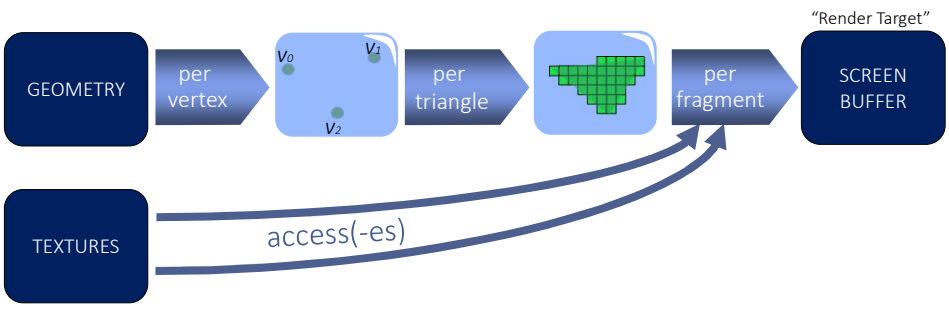
- Typically, lighting happens at the per fragment (per pixel) stage
  - the cheapest option, compute lighting per vertex, (and interpolate the resulting final RGB) saves computation but impacts quality (and disallows normal-maps and textures)

    formerly known as "Gouraud shading"
- Non uniform material parameters are
  - gathered from textures with texture accesses
  - or interpolated from per-vertex attributes (cheaper)

    heavily optimized, but still expensive
- Because lighting equations are now quite complex, this burdens the per-pixel stage considerably!
  - For this reason, games are often fill-limited

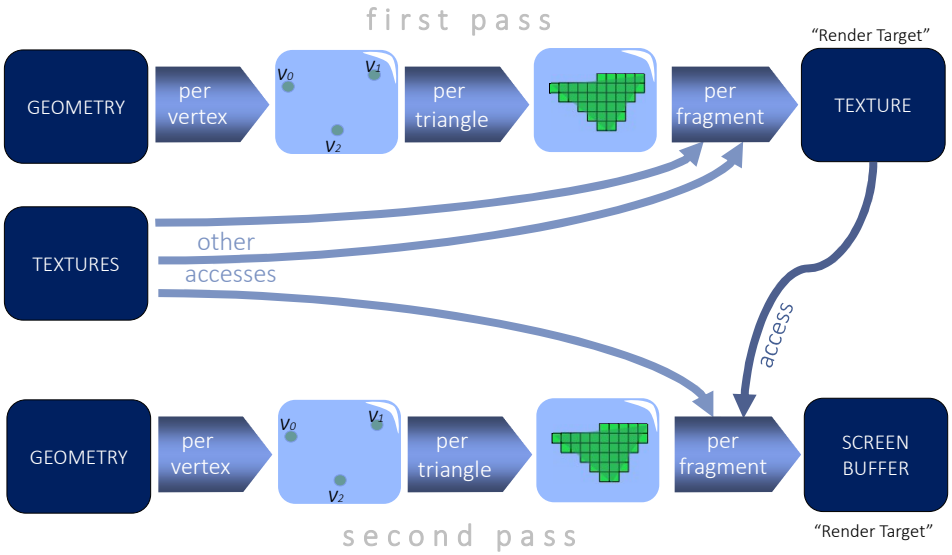
64

Texture access: it's in the per fragment process



65

basics: Render to Texture
(multi-pass rendering)



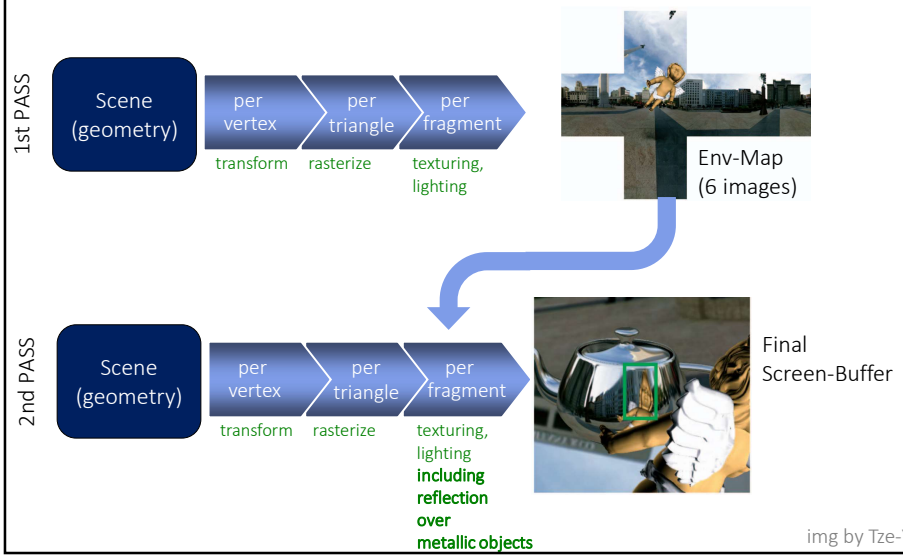
67

Multipass rendering techniques (general concept)

- 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"
  - normally, the render target is the "screen buffer" (the buffer shown to the screen)
  - this technique is aka "render to texture"
- 2nd pass: fill the final screen buffer
  - using the just-computed internal buffer as a 2D texture
- Note: efficient 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!

68

Example: metallic reflections of *dynamic* scenes



img by Tze-Yiu Ho

69

Main rendering algorithms: two classes of approaches

- Forward rendering
- Deferred shading

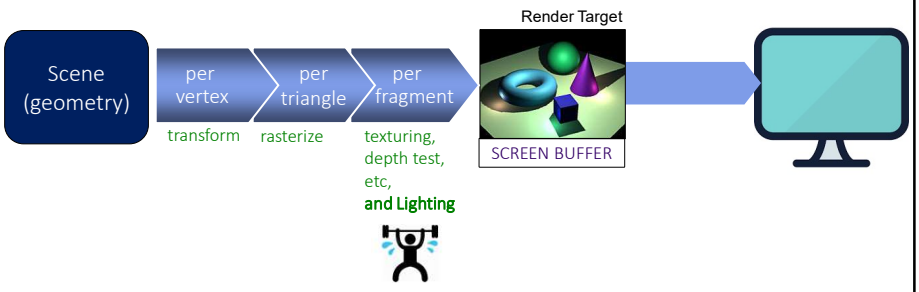
aka Deferred lighting (actually, a variation)
aka Deferred rendering (inappropriate?)

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

70

Main rendering algorithms: two classes of approaches

- Forward rendering

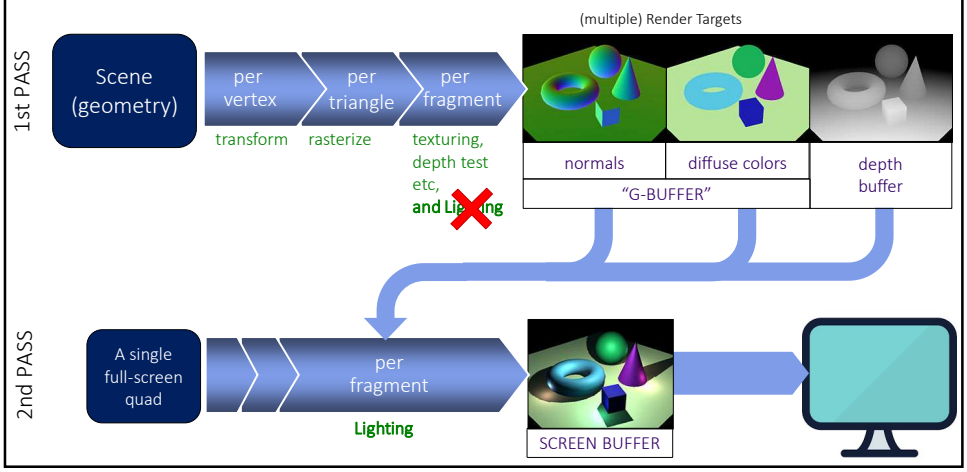


71

Main rendering algorithms: two classes of approaches

- Deferred shading

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



72

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

73

Ad-hoc rendering techniques popular in games: things we will see

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

with PCF

SSAO

DoF

HDR

NPR

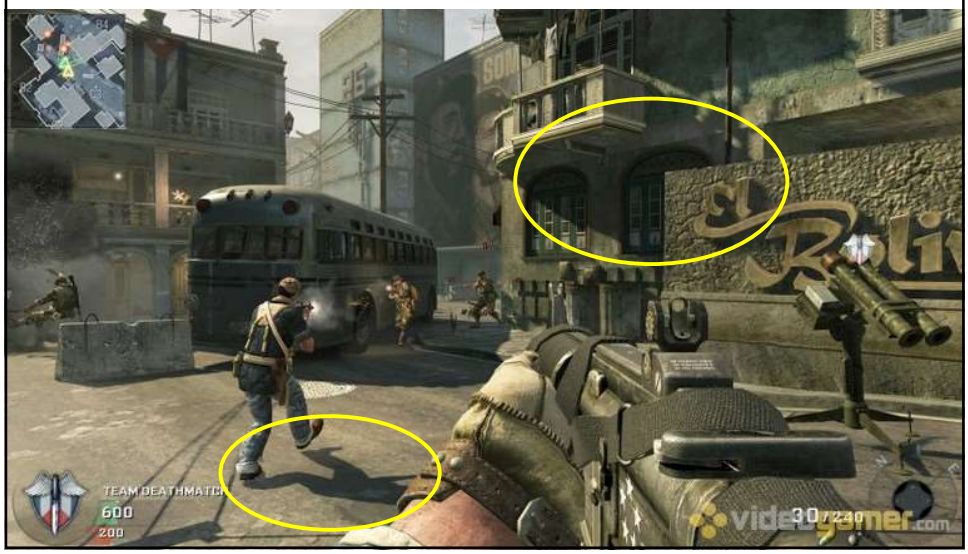
74

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

- 1st 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 buffer)
- 2nd pass:
  - render just one single "full screen" rectangle
  - (it filling the entire screens with two triangles)
  - for each produced fragment: apply 2D effects to the buffer
- Notes:
  - Basically, we can apply image filters to the rendering.
  - Many of the techniques in the previous slides are like this

76

Shadow mapping



78

Shadow mapping

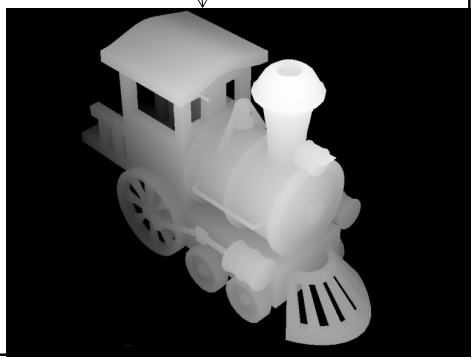
79

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)

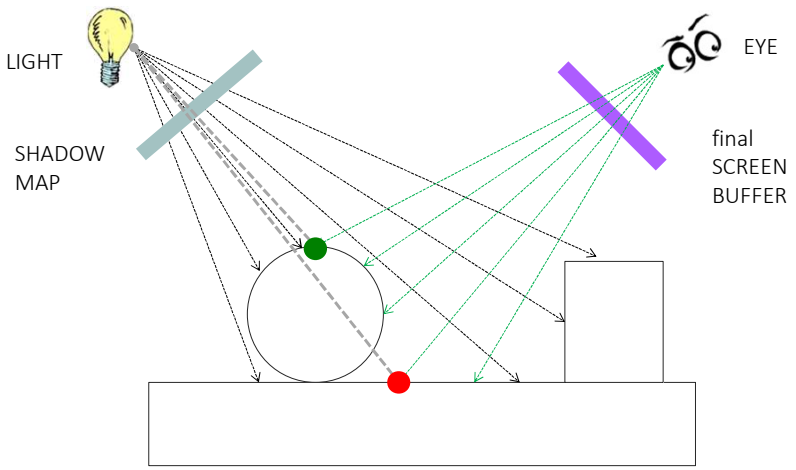
2nd pass:
- camera in final position
- for each fragment, access the shadow-map, determine if that if fragment is visible by light (or not)
- If not visible, negate contribution of that discrete light source

- Result:
  - Blockers cast a shadow



80

Shadow-mapping concept



81

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)

82

Shadow Mapping: effect of being in shadow

repeat for each light source

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

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

negated for that light source
(if with PCF: maybe only in part)

material parameter
light parameter
geometry

83

Shadow Mapping: effect of being in shadow

- Negates (zeroes) the light term of that (discrete) light-source
- Observe: the other light components are unaffected:
  - Other (non shadowed) lights
  - The ambient factor
  - Emission factor

84

Screen Space AO (SSAO)

SSAO only

85

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 multiples 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

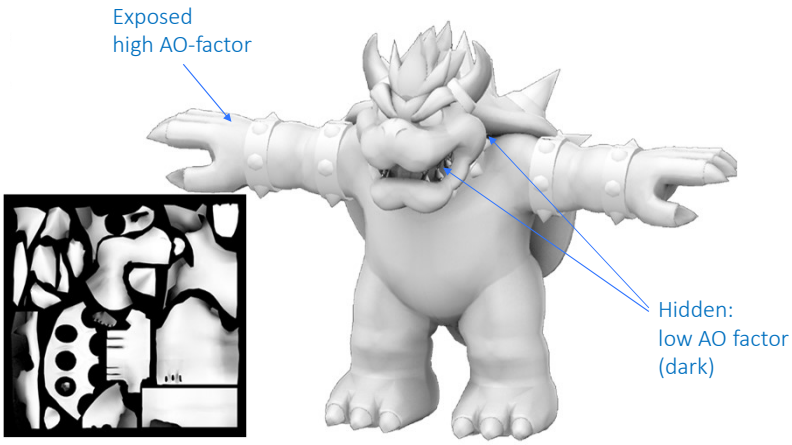
86

Two ways to compute AO: OSAO versus SSAO

- Object Space Ambient Occlusion (OSAO)
  - Baked in preprocessing on each mesh
  - Stored as a per-vertex attribute OR a texture (“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)
  - Screen space technique
  - 1st pass: compute depth map (maybe normal too)
  - 2nd 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
- Can be combined!

87

Baking AO over a mesh (OSAO)



Baked AO map

88

No SSAO

89

With SSAO

90

Screen Space AO in a nutshell

- 1st pass: standard rendering
  - produces: rgb image
  - produces: depth image
- 2nd pass:
  screen space technique
  - for each pixel, look at its depth VS depths of its neighbors:
    - Neighbors are in front?
      difficult to reach pixel: low AO factor (closer to 0)
    - neighbors are behind?
      pixel exposed to ambient light: high AO factor (closer to 1)

91

Ambient occlusion: effects

repeat for each light source

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

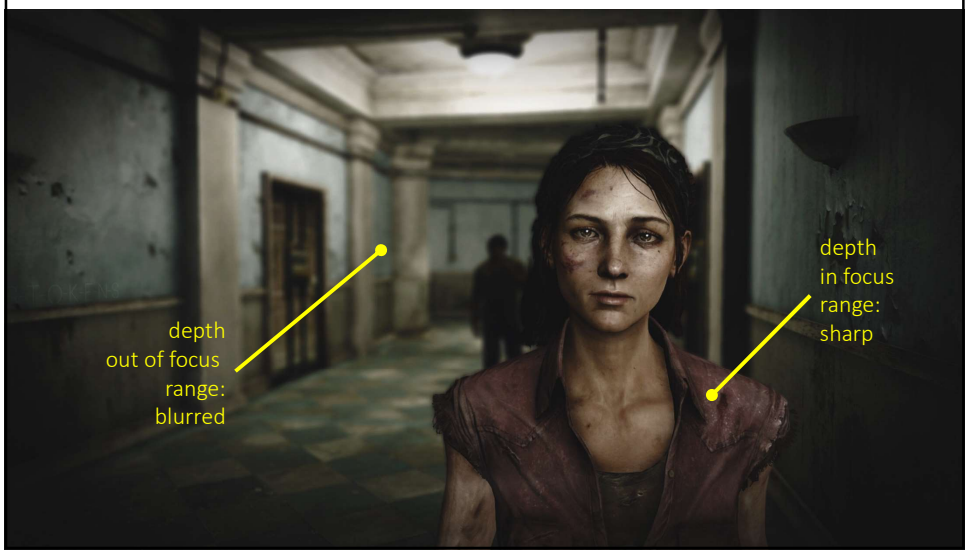
$$(\hat{n}\cdot\hat{L})\begin{pmatrix}d_R\\d_G\\d_B\end{pmatrix}\otimes\begin{pmatrix}L_R\\L_G\\L_B\end{pmatrix}+(\hat{n}\cdot\hat{H})^{E}\begin{pmatrix}s_R\\s_G\\s_B\end{pmatrix}\otimes\begin{pmatrix}L_R\\L_G\\L_B\end{pmatrix}+\cancel{\begin{pmatrix}a_R\\a_G\\a_B\end{pmatrix}\otimes\begin{pmatrix}A_R\\A_G\\A_B\end{pmatrix}}+\begin{pmatrix}e_R\\e_G\\e_B\end{pmat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

negates some % of this

material parameter
light parameter
geometry

92

(limited) Depth of Field



93

(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

94

HDR - High Dynamic Range (limited Dynamic Range)

95

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

96

Motion Blur

100

Non-PhotoRealistic Rendering (NPR)

- Any rendering technique not aimed at realism
- Instead, the objective can be:
  - imitating a given style (imitative rendering), such as:
    - cartoons ("toon shading") ← most popular!
    - pen-and-ink drawings
    - pencil sketches
    - pixel art ← popular in nostalgic retro games (niche)
    - manga, comics, etc ← very common
    - pastels, oil paintings, crayons …
  - clarity/readability (illustrative rendering)
    - usually not for games

101

Toon shading / Cel Shading

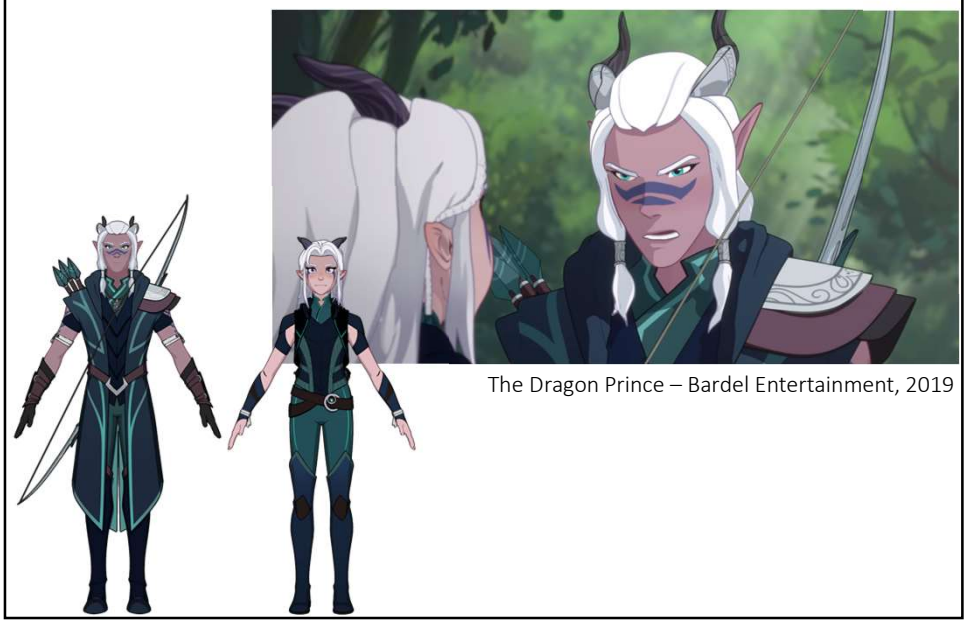
102

Toon shading / Cel Shading

(tweaked) Team Fortress II – Steam

103

Not just for games



The Dragon Prince – Bardel Entertainment, 2019

104

Toon shading / Cel Shading in a nutshell

- Simulating “toons” / hand drawn effect
- At its basics, a combination of two effects:
  - addition contour lines
    - lines appearing at discontinuities of:
      1. depth,
      2. normals,
      3. materials
  - quantized lighting:
    - e.g., 2 or 3 tones: light, medium, dark instead of continuous shades
    - a simple variation of lighting equation: quantize its result

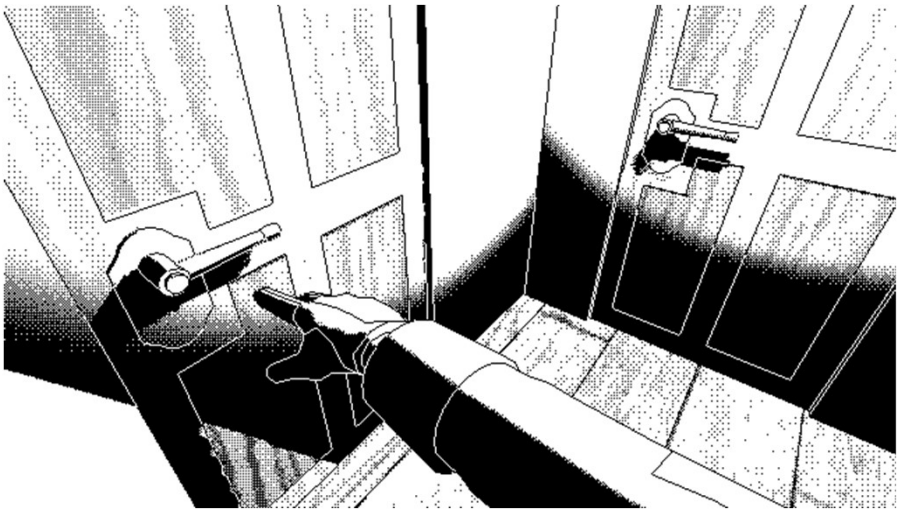
105

NPR rendering:
e.g.: simulated pixel art

img by Howard Day (2015)

106

NPR rendering:
simulated pixel art



img by Lucas Pope

108