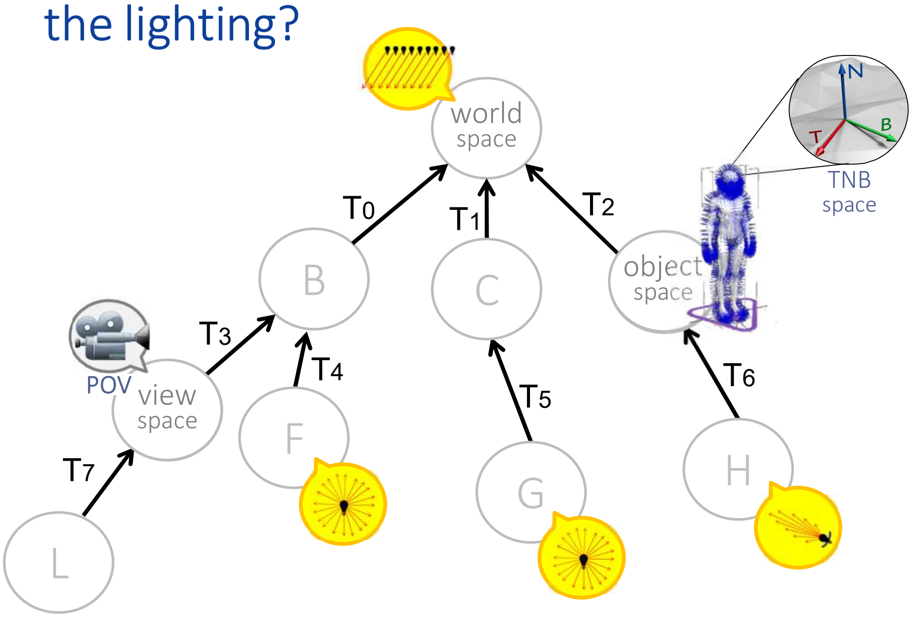
Course Plan

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. 8: Game 3D Animations
lec. 9: Game 3D Audio
lec. 10: Networking for 3D Games
lec. 11: Artificial Intelligence for 3D Games
lec. 12: Game 3D Rendering Techniques :-(

77

In which space to computue the lighting?



78

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, the most efficient solution is:
  - Use the same space the normals are expressed, in the texture
    For Tangent Space normal maps: the TBN space
  - 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
- In this lecture, we'll get a better understanding of the difference

for anisotropic materials

79

3D Videogames 2020/2021
Univ. degli Studi di Milano

Rendering in games

Part II: standard rendering algorithms for 3D games

80

Rendering in 3D games

- Real time
  - (20 o) 30 o 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

81

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

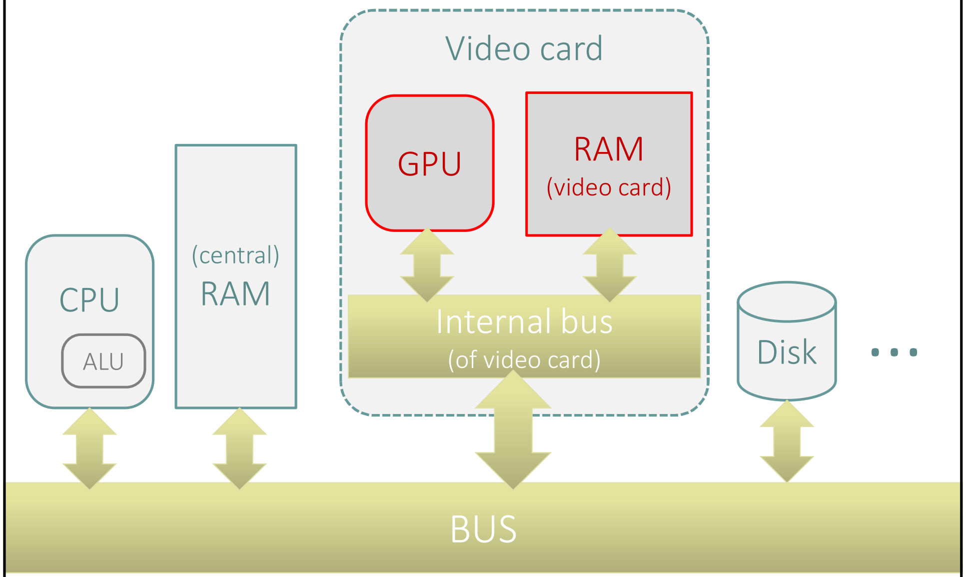
THE MESH ASSET

THE MATERIAL ASSET

In this lecture, we'll go lower level

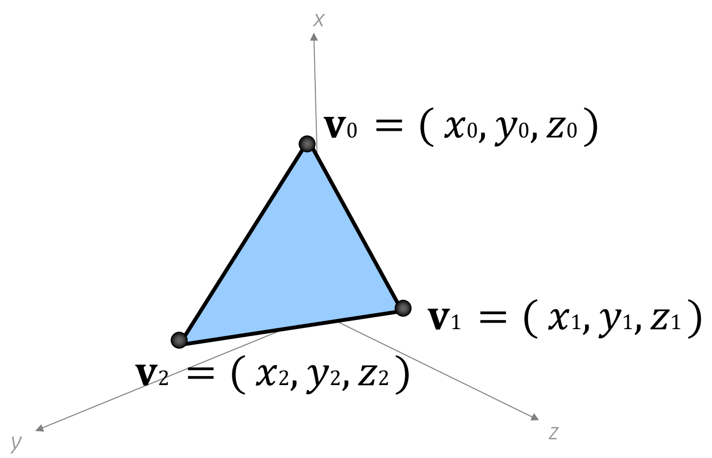
82

Graphic card



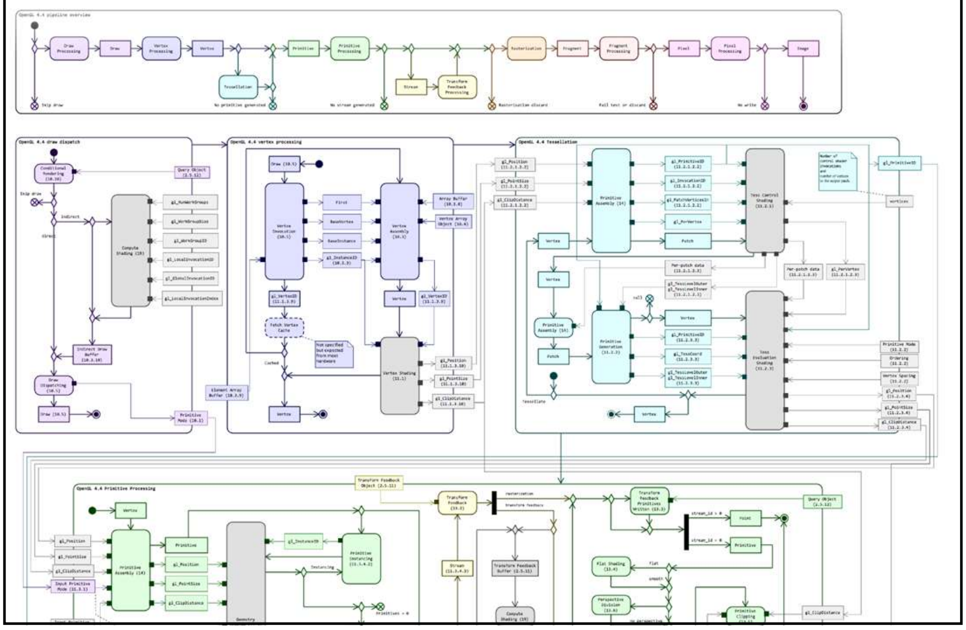
85

Rendering of a mesh = rasterization of all its triangles



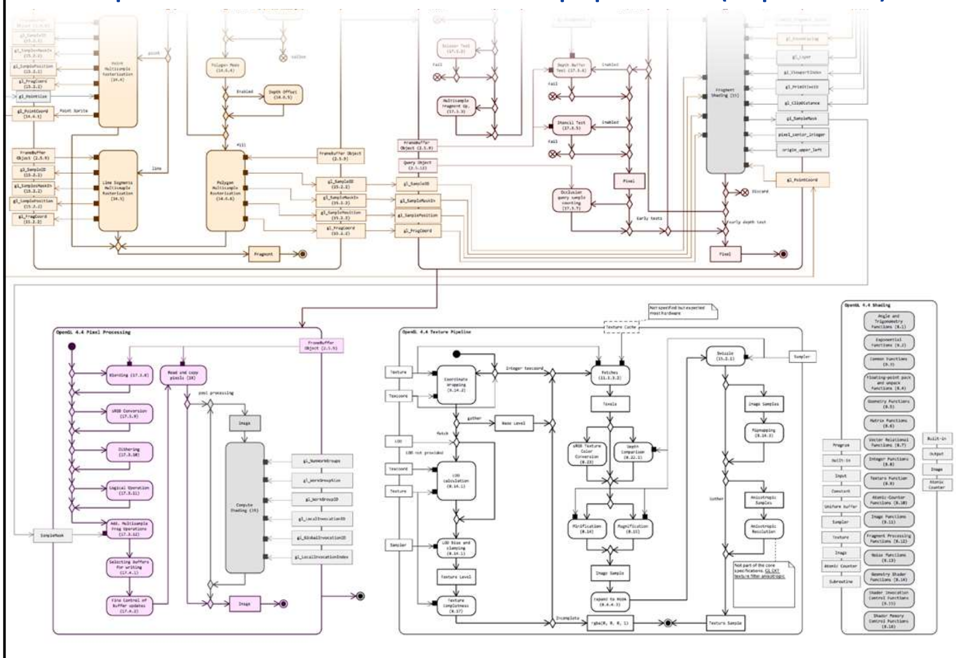
86

Example: full API for the GPU pipeline (OpenGL) 1/2



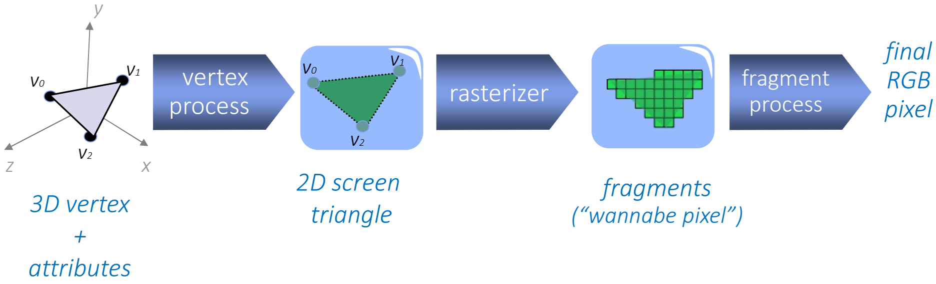
87

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



88

GPU pipeline – a simplified conceptual version



91

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,

92

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

93

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

95

GPU pipeline – bottlenecks
(remarks and terminology)

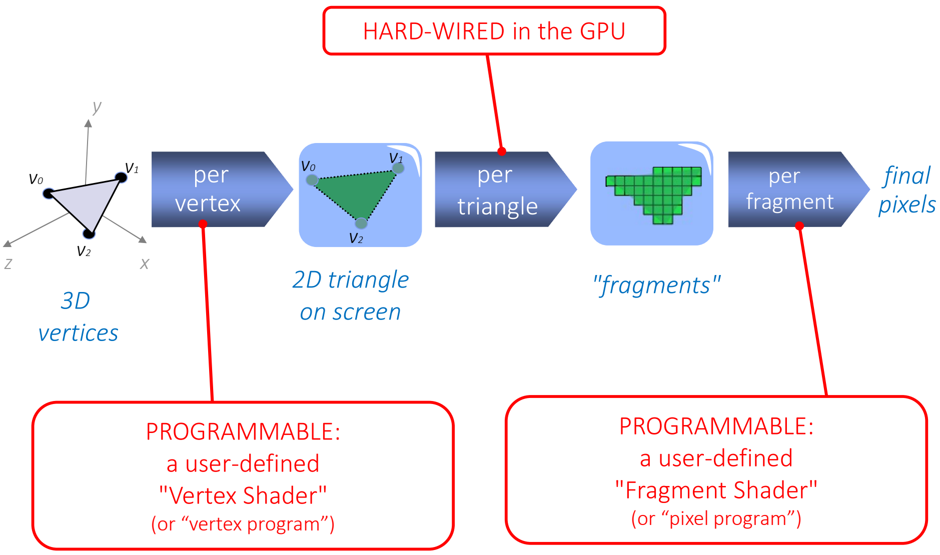
- Like in any pipeline, the process goes *as slow as its slowest stage*
  - i.e., the «bottleneck» of the pipeline determines the total speed
  - Any other stage is idle for part of the time (which is always a waste)
    - stages before the bottleneck are «chocked»
      (they cannot produce output because 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

96

Rasterization-Based Rendering



97

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

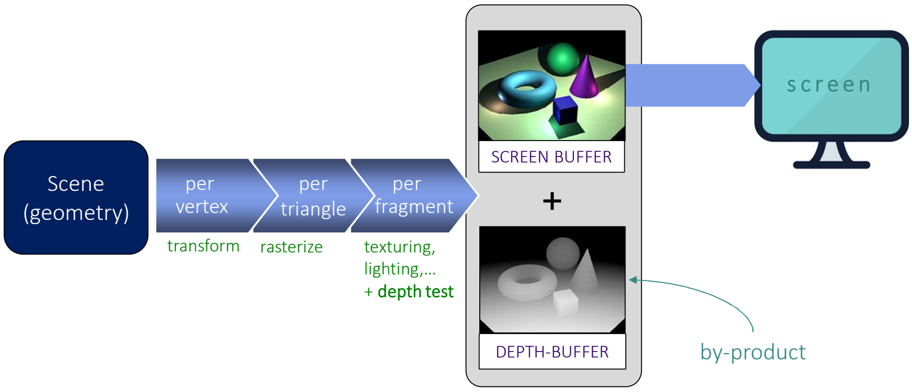
98

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)

99

basics: Depth buffer



100

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

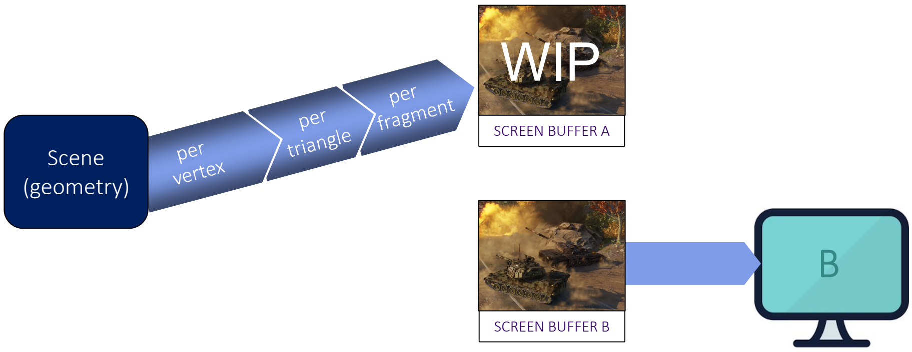
- Any rendering producing a screen-buffer ...
  - which is sent to the screen
- ...also produces a depth-buffer
  - as a by-product!
  - not set to the screen: it's an "offline" buffer
  - it's used during the rendering to determine occlusions and remove "hidden surfaces"
    (i.e. make what is behind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

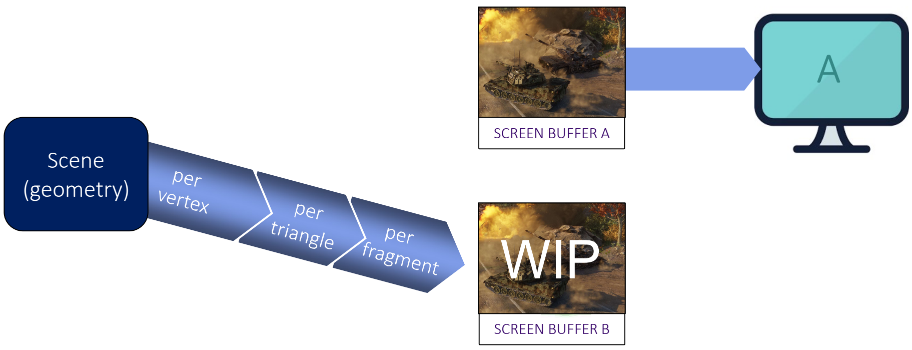
101

basics: Double Buffering



102

basics: Double Buffering



103

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

104

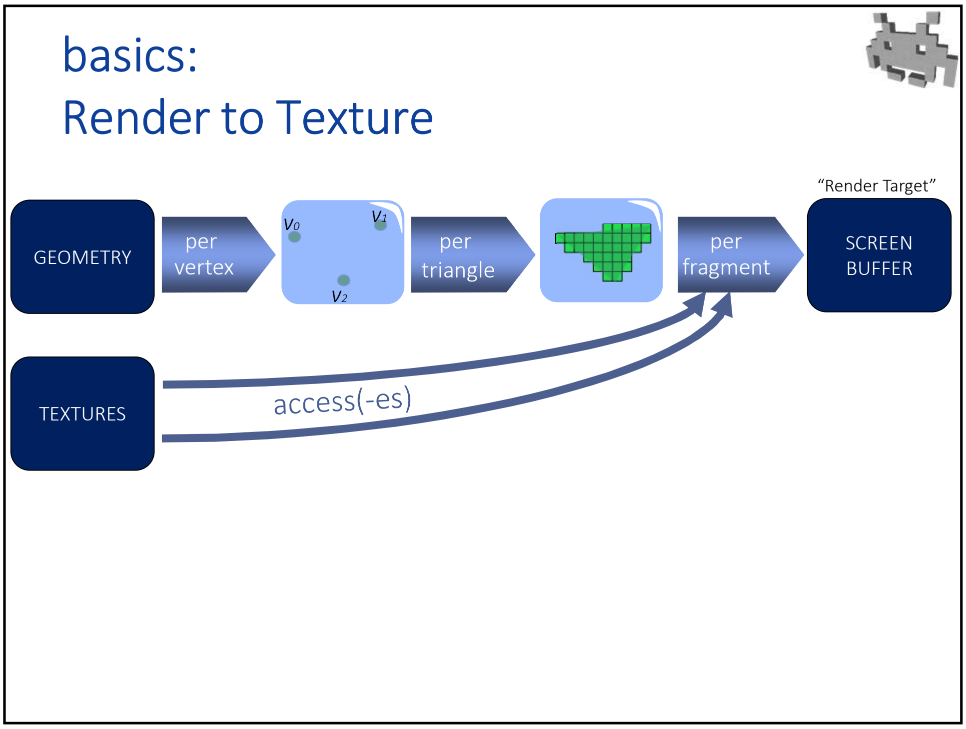
basics: Per-pixel lighting

- 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)

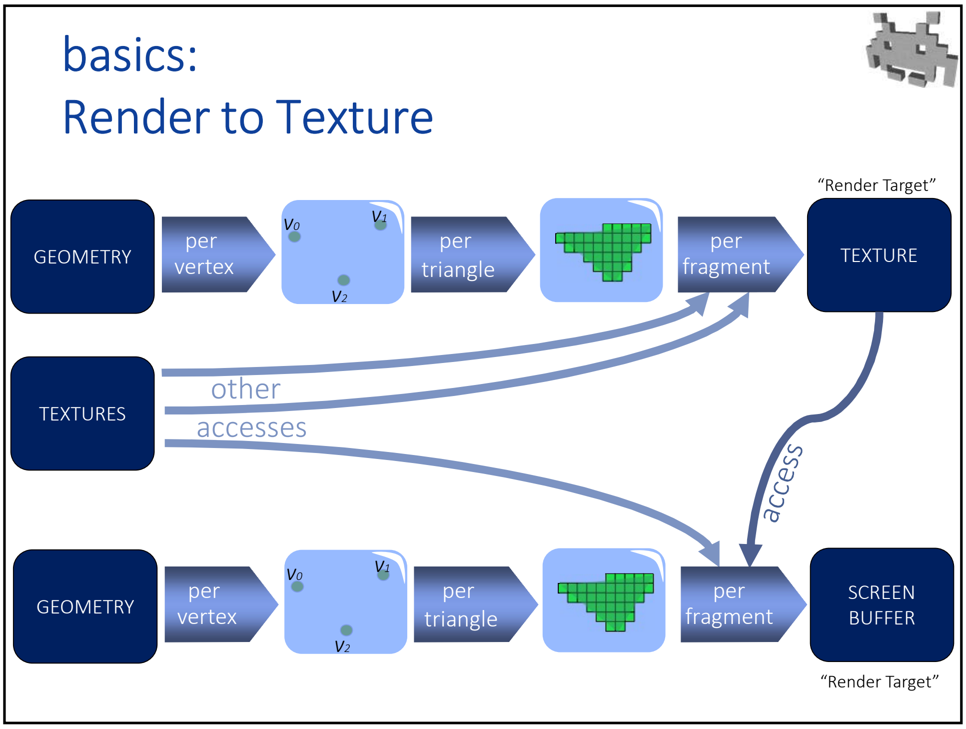
    ← formerly known as "Gouraud shading"
- Non uniform material parameters are
  - gathered from textures with texture accesses
  - or interpolated from per-vertex attributes (cheaper)

    ← 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

105


basics:
Render to Texture
"Render Target"
GEOMETRY
per vertex
v_0
v_1
v_2
per triangle
per fragment
SCREEN BUFFER
TEXTURES
access(-es)

106


basics:
Render to Texture
"Render Target"
GEOMETRY
per vertex
v_0
v_1
v_2
per triangle
per fragment
TEXTURE
TEXTURES
other
accesses
access
GEOMETRY
per vertex
per triangle
per fragment
SCREEN BUFFER
"Render Target"

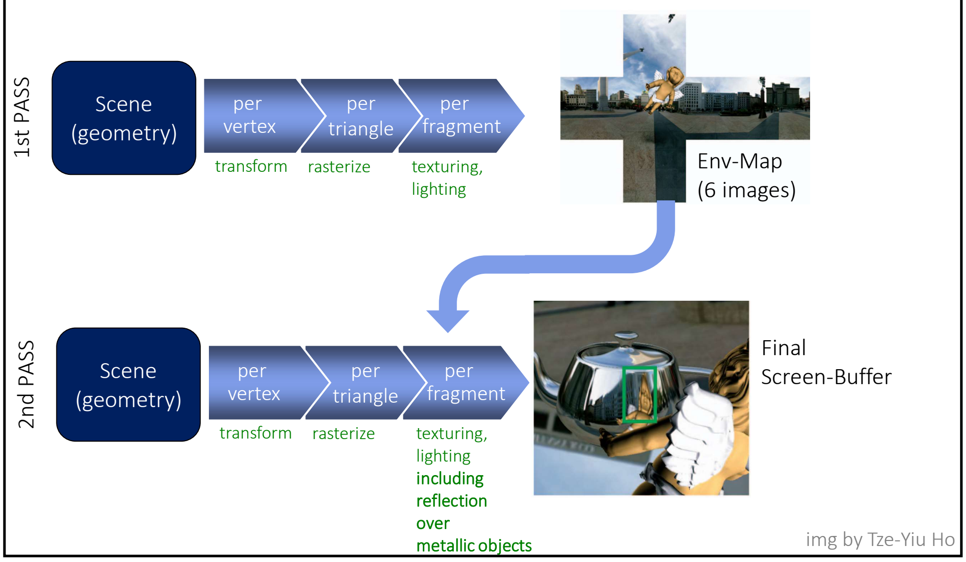
108

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!

109

Example: metallic reflections of *dynamic* scenes



img by Tze-Yiu Ho

110

Marco Tarini
Università degli Studi di Milano

Main rendering algorithms: two classes of approaches

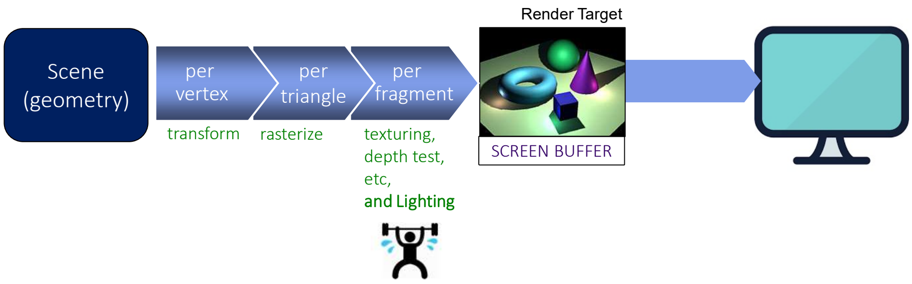
- Forward rendering
- Deferred shading ← aka Deferred lighting (actually, a variation) / aka Deferred rendering (inappropriate?)

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

111

Main rendering algorithms: two classes of approaches

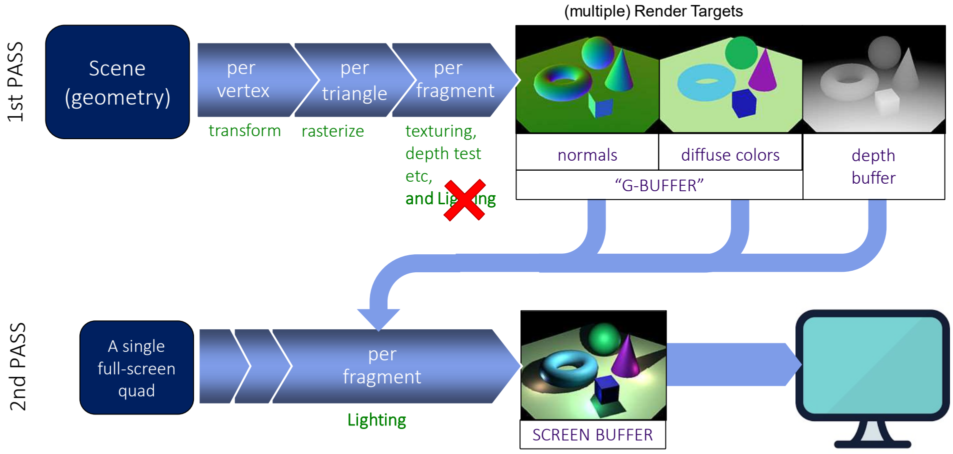
- Forward rendering



112

Main rendering algorithms: two classes of approaches

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



113

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

114

Ad-hoc rendering techniques popular in games: a summary

- 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

115

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

117

Shadow mapping

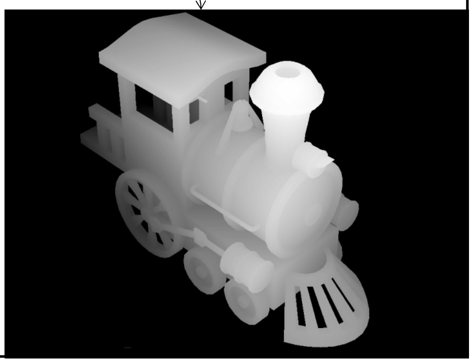
120

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)

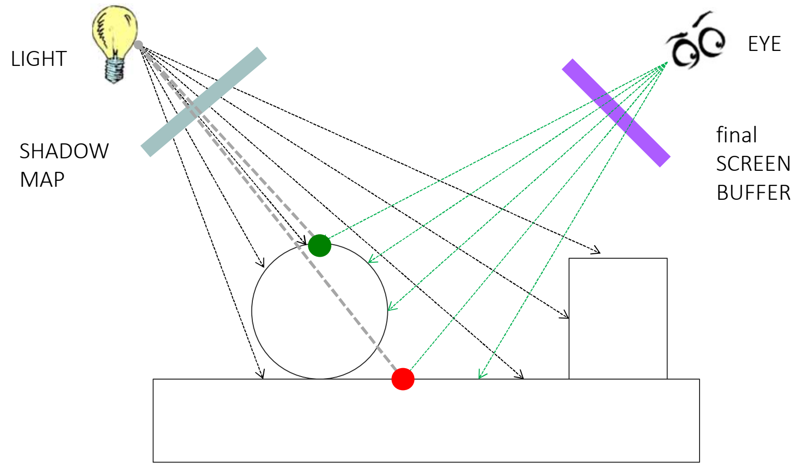
2nd pass:
- camera in final position
- for each fragment, access the shadow-map, determine if that if fragment is visible by light (or not)
- If not visible, negate contribution of that discrete light source

- Result:
  - Blockers cast a shadow



121

Shadow-mapping concept



122

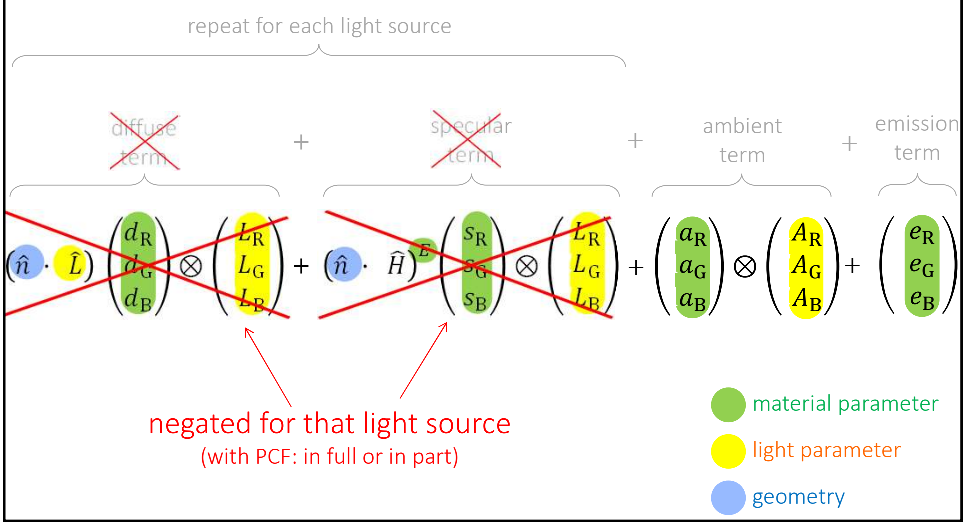
Shadow mapping: issues

- 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)

123

Shadow Mapping: effect of being in shadow



124

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

125

Screen Space AO (SSAO)

SSAO only

126

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

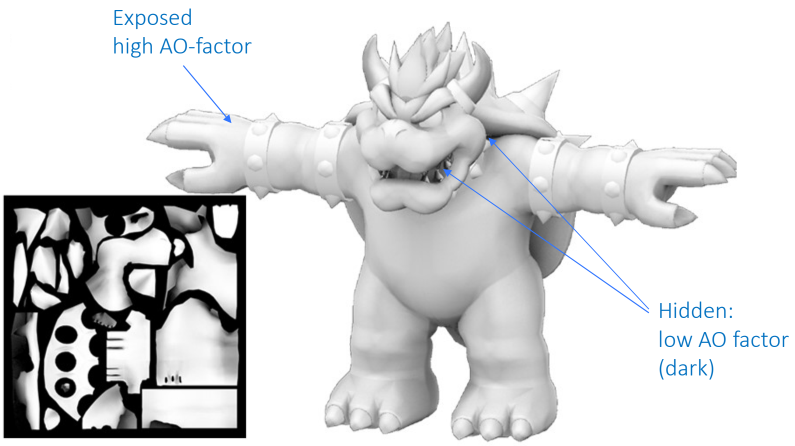
127

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!

128

Baking AO over a mesh (OSAO)



Baked AO map

129

No SSAO

130

With SSAO

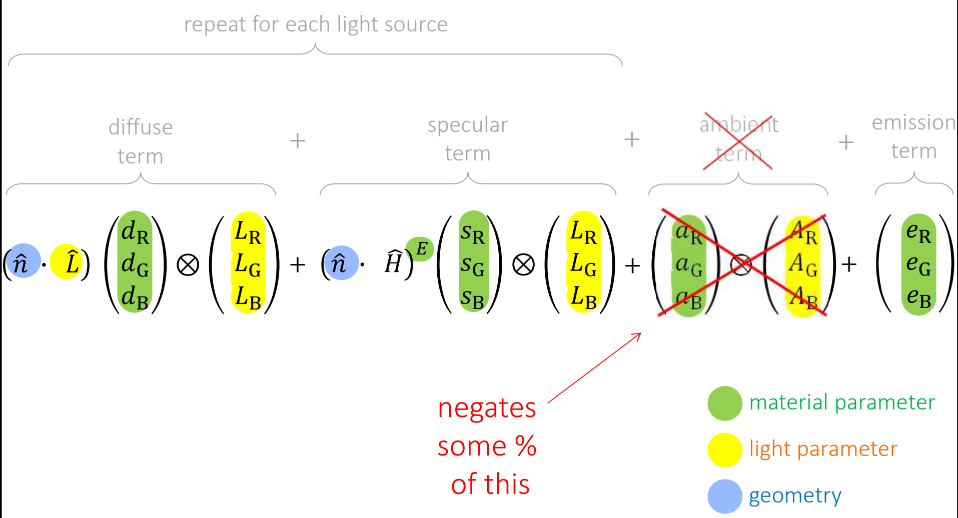
131

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 its neighbor depths:
    - Neighbors are in front?
      difficult to reach pixel: darken ambient
    - neighbors are behind?
      pixel exposed to ambient light: keep it lit

132

Ambient occlusion: effects



133

(limited)
Depth of Field

134

(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

135

HDR - High Dynamic Range (limited Dynamic Range)

136

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

137

Parallax mapping: in a nutshell

- Texture-for-geometry technique
- Texture used:
  - displacement maps
  - color / rgb map

138

Parallax Mapping

139

Parallax Mapping

140

Motion Blur

141

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

142

Toon shading / Cel Shading

143

Toon shading / Cel Shading

(tweaked) Team Fortress II – Steam

144

Not just for games

The Dragon Prince – Bardel Entertainment, 2019

145

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

146

NPR rendering: e.g.: simulated pixel art

img by Howard Day (2015)

147