Deep G-Buffers for stable Global Illumination Approximation

Ferit Tohidi Far

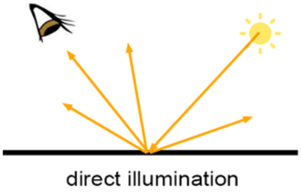
February 25, 2019

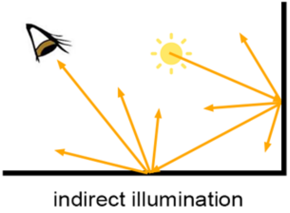
Content

- Global illumination
  - Pathtracing
  - Radiosity
  - Visual effects
  - Computational difficulty
- Traditional rendering
  - Forward rendering
  - Deferred rendering
- Deep G-Buffers
  - Depth-peeling
  - Generating a 2-layer deep g-buffer
  - Global illumination approximation

Global illumination

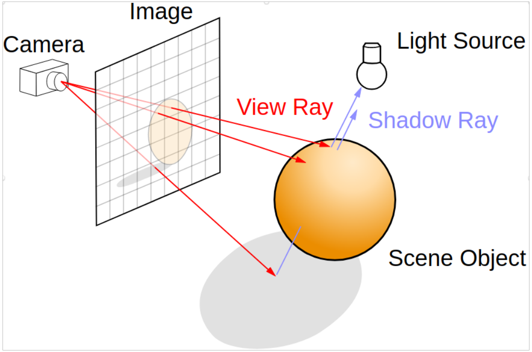
- lighting of a scene
- direct **and indirect** light is considered
- causes visual effects that convey realism
- $L_o(\omega) = L_e(\omega) + \int_\Omega f(\omega,\omega')L_i(\omega')cos(n,\omega')\partial\omega'$
- most popular method is pathtracing





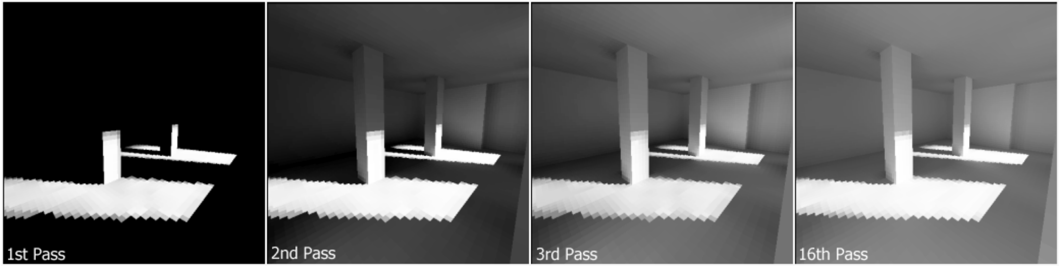
Pathtracing

- send camera ray through each pixel
- allow ray to reflect **diffusely or specularly**
- trace it back to a light source
- if a light source was hit, the pixel is colored
- else the pixel is black
- sample each pixel thousands of times, then average

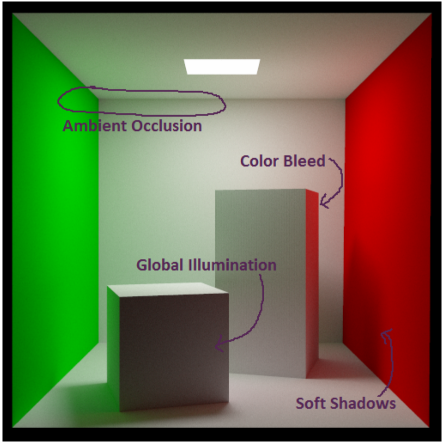


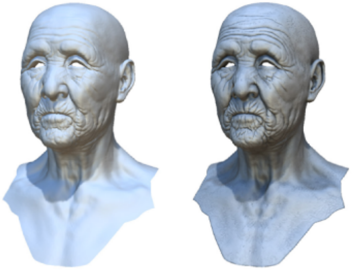
Radiosity

- scene is divided into patches
- each patch is a light receiver and emitter
- initialize scene with at least one patch that emits non-zero amount of light
- iteratively update receivance and emitance of each patch
- **purely diffuse global illumination**



Visual effects





Original model

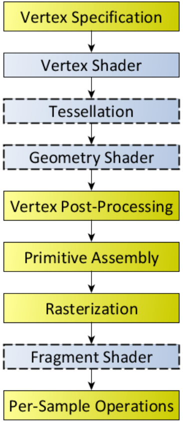
With ambient occlusion

Computational difficulty

- pathtracing
  - computes multiple ray bounces
  - requires (hundreds of) thousands of samples
  - has to be recomputed if camera or object moves
- radiosity
  - needs large amount of patches for good results
  - has to be recomputed if an object moves
- tough to be computed in real-time without additional tricks

Traditional rendering

- rasterization
- way easier to compute than raytracing
- interactive framerates
- trade-off: not as realistic
- requires techniques to simulate visual effects
- most of the rendering is done by GPU
- abide by rendering pipeline of GPU

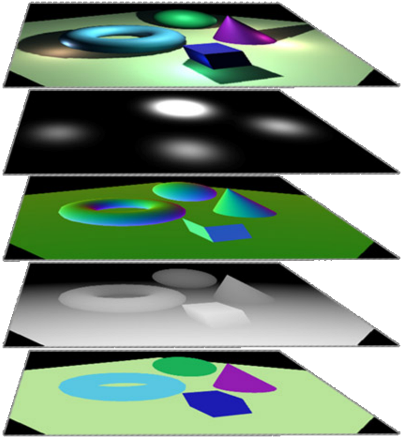


Forward rendering

- computes geometry and lighting in a single pass:
  - for each fragment compute lighting
  - do z-test (closest fragment for that xy-position?)
  - if passed, render to frame-buffer, else discard
  - render frame-buffer to screen
- lighting computed regardless if fragment visible or not
- however, benefitial for transparency and anti-aliasing

Deferred rendering

- computes geometry in first pass:
  - albedo-buffer
  - normal-buffer
  - z-buffer
- render g-buffers to texture-buffer (not screen)
- compute lighting in second pass:
  - read frontmost geometry from g-buffer
  - compute lighting
  - render to screen
- only computes lighting for visible fragments



Benefits of deferred rendering

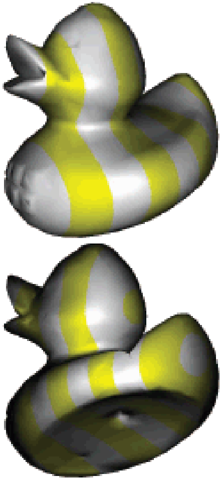
- forward rendering in $O(objects \cdot lights)$:
  - for each object
    - for each light compute lighting
- deferred rendering in $O(objects + lights)$:
  - for each object
    - render geometry
  - for each light
    - compute lighting
- lighting computation is way less complex!
- we can add way more lights to the scene

Problems of deferred rendering

- we **only** store information about frontmost fragments
- but information of deeper layers can improve visual effects
- get g-buffers of deeper layers through depth-peeling!

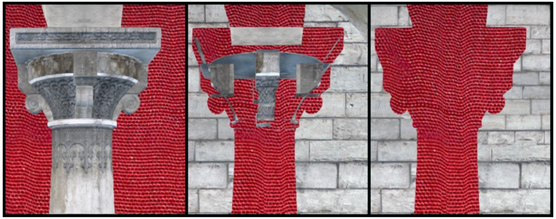
Depth-peeling

- compute first layer g-buffers as usual
- for further layers, peel away the previous layers
- effectively returns closest-, second-closest, ... n-closest layers



Deep G-Buffers

- idea: also gather geometry of n-closest layer
- generate 2-layer deep g-buffer with depth-peeling or oracle
- enforce minimum depth separation
- consider second layer for visual effects



Primary Traditional Peeling Minimum Separation

Generating a 2-layer deep g-buffer (depth-peeling)

- **depth-peeling method**
- collect first layer g-buffer as usual
- compute second layer g-buffer by peeling the first layer
- takes two passes over scene geometry

Generating a 2-layer deep g-buffer (depth-peeling)

- **depth-peeling method**
- collect first layer g-buffer as usual
- compute second layer g-buffer by peeling the first layer
- takes two passes over scene geometry
- instead use oracle and do it in a single pass!

Generating a 2-layer deep g-buffer

- **use some oracle to predict first layer z-buffer**
- remember that we are running some simulation/animation
- frames are computed per time-step
- after a time-step, the object locations won't change **that much**
- we even have knowledge of position and velocity updates
- exploit this by recycling information from the previous frame and adjusting it a little
- 4 different variants available

Generating a 2-layer deep g-buffer (previous variant)

- **previous variant**
- recycle first layer z-buffer of previous frame
- the smaller the position updates, the smaller the error
- even then, errors would only appear in the second layer (invisible unless transparent)
- does not guarantee minimum separation

Generating a 2-layer deep g-buffer (delay variant)

- **delay variant**
- introduce a frame of latency
- frame and animation/simulation are out of sync by one frame
- use first layer z-buffer of precomputed latency frame
- drawback: one frame of latency

Generating a 2-layer deep g-buffer (predict variant)

- **predict variant**
- use velocities from animation/simulation
- predict position updates of objects
-

Generating a 2-layer deep g-buffer (reproject variant)

- **reproject variant**
- performs minimum separation test against previous frame's first layer z-buffer
- visibility test done using previous z-buffer ("in the past")
- same source of error as predict variant, but not as bad (velocities are perfect)
- delivers most stables performance out of the 4 variants

Stable global illumination approximation

- compute ambient occlusion
- compute some radiosity steps
- compute screen space reflections
- apply direct and ambient light

Screen space ambient occlusion

- screen space ambient occlusion SSAO
- compute ambient occlusion factor AO for both layers
- depth discontinuity is now accouted for
- results will look more like actual ambient occlusion instead of approximate

Screen space radiosity

- divide screen space into patches using g-buffers
- define form-factor taking into account both g-buffer layers
- compute **one** bounce per frame
- accumulate bounces from previous frames
- ghosting may occur: introduce ways to damp previous lighting

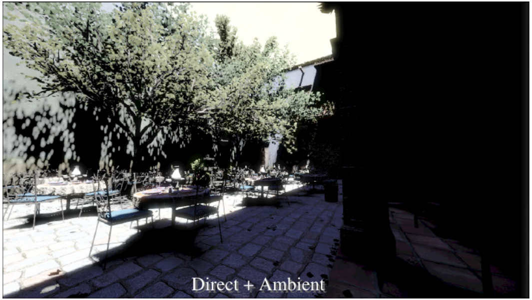
Screen space mirror reflections

- march reflection rays in cameraspace
- project reflection onto both g-buffers
- if distance is within minimum depth separation, it's a hit
- else switch to cube-mapping

Screen space mirror reflections (cube mapping)

- render scene from 6 angles (cube-like)
- march reflection ray
- look-up reflection point in cube-map

Results



- 1920x1080 resolution
- rendered using NVIDIA GeForce 980
- in 10.8ms (92 FPS)
- looks good