Notes

Alpha

- We add another channel, alpha: RGBA
- Encodes whether the pixel of the image is empty (alpha=0) or opaque (alpha=1) or something in between (0<alpha<1)
  - Most important case: at the edges of objects
- When we render a layer, we compute and save alpha along with RGB
  - Or if it's real action, use a "blue screen" behind the actors
- Premultiplied alpha: instead of storing regular RGB + alpha, store
  rgb=alpha*R, alpha*G, alpha*B
  - Simplifies formulas to come

Atop operation

- Image 1 "atop" image 2
- Assume independence of sub-pixel structure
  - So for each final pixel, a fraction $alpha_1$ is covered by image 1
  - Rest of final pixel (a fraction of $1\text{-}alpha_1$) is covered partly by image 2 (fraction $alpha_2$) and partly uncovered
- Without premultiplied alpha:
  - $R_{final}=alpha_1*R_1 + (1\text{-}alpha_1)*alpha_2*R_2$
  - $G_{final}=alpha_1*G_1 + (1\text{-}alpha_1)*alpha_2*G_2$
  - $B_{final}=alpha_1*B_1 + (1\text{-}alpha_1)*alpha_2*B_2$
  - $alpha_{final}=alpha_1 + (1\text{-}alpha_1)*alpha_2$

Premultiplied

- Using standard premultiplied alpha, formulas simplify:
  - $R_{final}=r_1 + (1\text{-}alpha_1)*r_2$
  - $G_{final}=g_1 + (1\text{-}alpha_1)*g_2$
  - $B_{final}=b_1 + (1\text{-}alpha_1)*b_2$
  - $alpha_{final}=alpha_1 + (1\text{-}alpha_1)*alpha_2$
- And of course store the result premultiplied:
  - $r_{final}=alpha_{final}*R_{final}$
  - $g_{final}=alpha_{final}*G_{final}$
  - $b_{final}=alpha_{final}*B_{final}$

Note on gamma

- Recall gamma: how nonlinear a particular display is
  - When you send a signal for fraction x of full brightness, actual brightness output from display is a nonlinear function of x
    - Called gamma since usually modeled as x^γ
  - For final image, for a particular display, should correct for gamma
- But when we're taking linear combinations of RGB values, need to do it before gamma correction!
  - Similarly for real life elements, camera output is distorted, needs to be undone before compositing

Sampling and Filtering

- For high quality images need to do
  - Antialiasing - no jaggies
  - Motion blur - no strobing
  - Possibly depth-of-field - no pinhole camera
- Boils down to:
  - Each pixel gets light from a number of different objects, places, times
- Figuring out where: point sampling
  - Find light at a particular place in the pixel, at a particular time, ...
- Combining the nearby point samples into RGBA for each pixel: filtering
  - Simplest is box filter (average the samples in pixel)

How to get point samples

- Three big rendering algorithms
  - Z-buffer / scanline
    - Graphics Hardware - OpenGL etc.
  - Ray tracing
    - Highly accurate rendering
    - Difficult models (e.g. volumetric stuff)
  - REYES
    - Almost everything you see in film/TV

REYES

- Invented at Lucasfilm (later Pixar) by Cook et al. SIGGRAPH '87
- Geometry is diced up into grids of micropolygons (quads about one pixel big)
- Each micropolygon is "shaded" in parallel to get a colour+opacity (RGBA)
- Then sent to "hiding" to determine in which point samples it makes a contribution
- Each point sample keeps a sorted list of visible points, composites them together when done
- Filter blends point samples to get final pixels

Why REYES

- No compromises in visual quality (antialiasing, motion blur, transparency, etc.) compared with e.g. OpenGL
- Very efficient in optimized implementations (e.g. PhotoRealistic Renderman from Pixar) with predictable runtimes compared to raytracing
  - Handle complex scenes robustly
- Huge flexibility from shading architecture
  - Modern GPU's are catching up now...

Shading

- A shader is a small program that computes the RGBA of a micropolygon
- Writing shaders to get the right look is a hugely important part of production
- Shaders probably need to know about surface normal, surface texture coordinates, active lights, ...
  - E.g. ubiquitous Phong model
- But can do a whole lot more!

Displacement shaders

- Can actually move the micropolygon to a new location
  - Allows for simple geometry (e.g. sphere) to produce complex results (e.g. baseball)
  - A flexibility no other renderer allows so easily
- Also could perturb surface normal independently (bump mapping)---so even if geometry remains simple and fixed, appearance can be complex

Shadows and lighting

- Part of the shader will figure out if point is in shadow or not
  - Typically using precomputed "shadow maps" but could also use ray-tracing!
- Huge flexibility for cheating ("photosurrealism")
  - Each object can pick and choose which lights illuminate it, which shadows it's in, change the location/direction/intensity of lights, ...
  - Way more control than e.g. ray-tracing

Surface shading

- Implement whatever formula you want for how light bounces off surface to viewer
- Can look up texture maps or compute procedural textures
- Can layer surface shaders on top of each other
- Reflections: can do ray-tracing for high accuracy, but usually use "environment mapping"
  - Assume object is small compared to distance to surroundings
  - Then look up reflected light in a texture based on direction (2D), not position+direction (5D)
  - Can capture or synthesize environment map

Atmospheric shading

- Can further adjust micropolygon colour to account for fog, atmospheric attenuation (blue mountains), etc.
- Could do it with more accurate ray-tracing
- Usually a simple formula based on distance to camera works fine

RenderMan

- Pixar defines a standard API for high quality renderers
  - Think OpenGL, but with quality trumping performance
  - "Postscript for 3D"
- Today, many software packages implement (at least part of) the Renderman standard
- Two parts to the API
  - Direct calls, e.g. RiTranslate(0.3, 0.4, -1);
    - Very similar to OpenGL
  - RenderMan Interface Bytestream (RIB) files: save calls in text file for later processing
- Also a shading language (based on C)

RenderMan resources

- Pixar website has official API reference
- "The RenderMan Companion", Upstill
- "Advanced RenderMan", Apodaca and Gritz
- SIGGRAPH course notes
- www.renderman.org