Ray Tracing + Cel Shading Renderer

Philip Carr and Thomas Leing
CS 174

Overview

Project Goals:

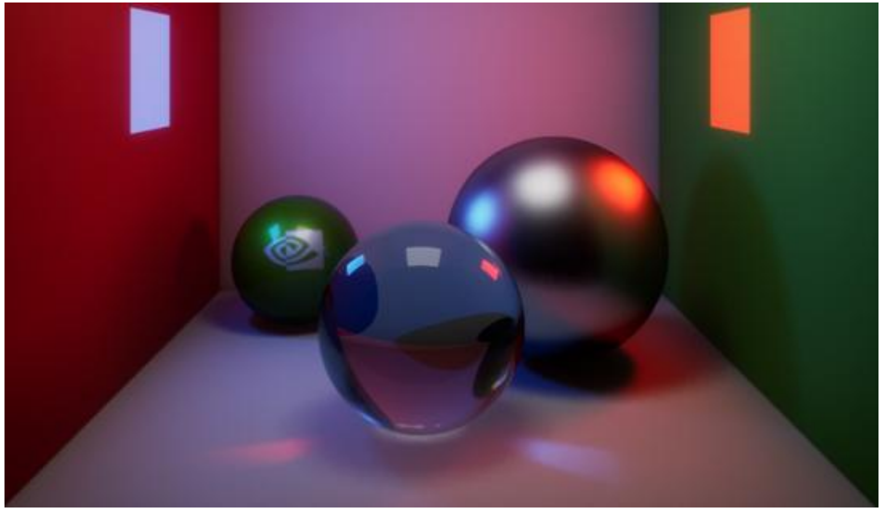
- Produce a rendering engine that was capable of the following:
    - CPU and GPU-accelerated (CUDA) ray tracing (non-OpenGL GPU implementation done for the CS 179 project)
    - CPU and GPU-accelerated (GLSL) cel shading
    - Water wave simulation
- Due to limited time, water wave simulation was attempted, but not completed

Introduction: Ray Tracing

- Form of photorealistic rendering
- Simulates the way in which light would naturally illuminate the scene, as perceived by the camera/viewer
- Send rays out from camera direction/position into scene
  - Each pixel corresponds to a ray sent into the scene in the direction of the pixel in ndc space
  - Detect intersection between ray and some location on an object
  - If intersection occurs, color of point determined using Phong lighting model
  - Shadows implemented by sending ray from intersection location to light source and determining whether this ray intersects another location of some object along the way
- KDTree data structure used to dramatically speed up ray-object intersections, especially when ray tracing with triangle meshes
  - KDTree spatially organizes triangle mesh for logarithmic-time speedup

Ray Tracing Examples

Nvidia RTX demos



https://www.awn.com/news/nvidia-unveils-quadro-rtx-worlds-first-ray-tracing-gpu

https://devblogs.nvidia.com/ray-tracing-games-nvidia-rtx/

Ray Tracing Algorithm Overview

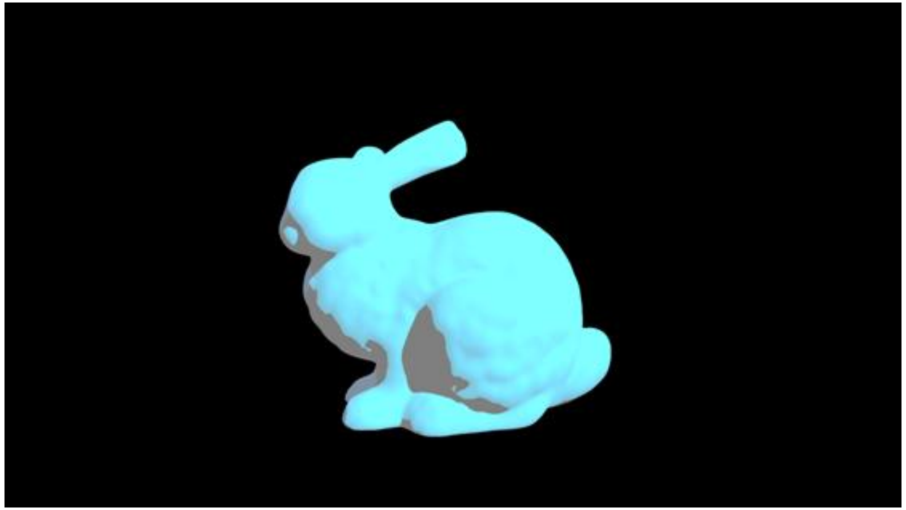
- Produce a KDTree for every triangle mesh object in the scene
- Given XRES x YRES screen resolution, loop over all (x, y) coordinates of the screen
    - For each pixel at location (x, y), produce a ray corresponding to the ndc point on the screen - camera position as a vector subtraction
    - Traverse the KDTree to detect whether this ray intersects any triangle mesh object
    - If no intersection is detected, color of pixel is set to black
    - If an intersection occurs:
        - For each light source in the scene, create a ray starting at the light source directed at the intersection point, and test to see whether an intersection occurs between the light source and the original intersection, which would cast a shadow
        - Use the phong lighting model to determine the color of the point using the object's ambient color, as well as the diffuse and specular colors produced with light sources that the intersection point is directly exposed to

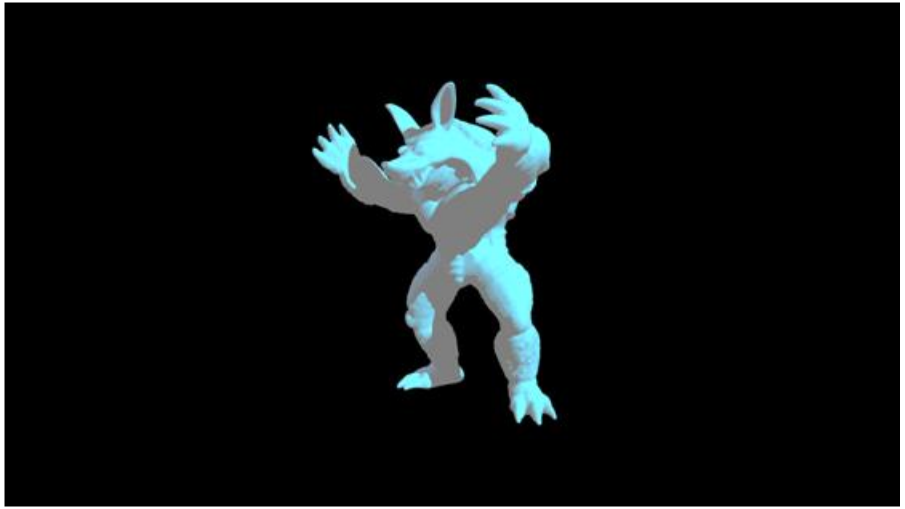
Ray Tracing Implementation Details and Results

- Ray tracing code built off of CS 171 assignments 5 and 7 (and assignment 2 to some extent)
- Major difference between CPU and GPU implementations was primarily just where in memory of the KD Trees were located (on CUDA device for GPU)
- GPU implementation computed ray-object intersections on the GPU for all rays given to the device (such as all the camera rays needed to render the image)
- Due to currently unknown CUDA memory issues, GPU implementation capped at maximum size of roughly 150x150-resolution images.
- GPU implementation runs approximately as fast as CPU implementation (both on Titan)

Ray Tracing Visual Results

CPU Results (4K images produced using Titan)





Ray Tracing Visual Results (Continued)

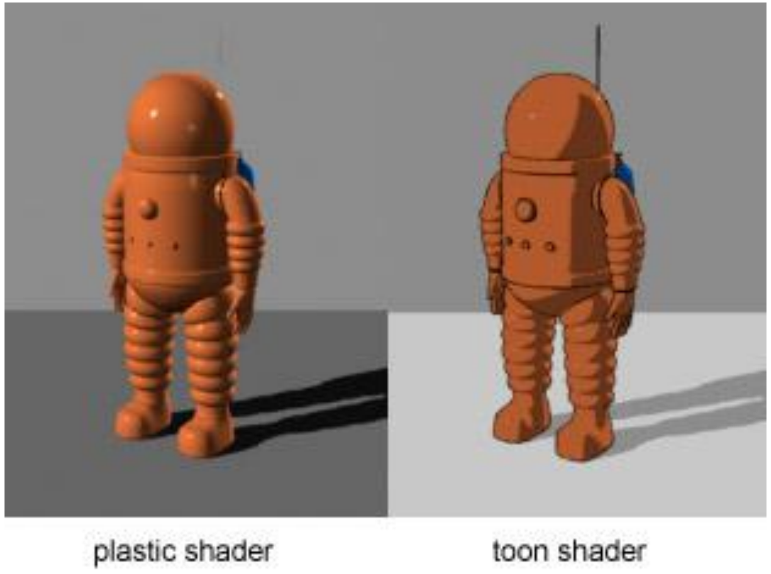
GPU (CUDA) Results (also run on Titan)

Introduction: Cel Shading

- Form of non-photorealistic rendering (NPR), domain of computer graphics focused on produce different aesthetics that diverge from standard photorealism
  - Cel shading (aka toon shading) designed to render 3D objects to appear somewhat “flat” on screen, with a visual style similar to that of a cartoon or comic book
- Two major components that define the visual style of cel shading
  - Easily noticeable object outlines
  - simplified , discretized color gradient of the object
    - On surface of object, colors transition from lighter to darker in discrete steps, rather than continuously (e.g. using the Phong lighting model)

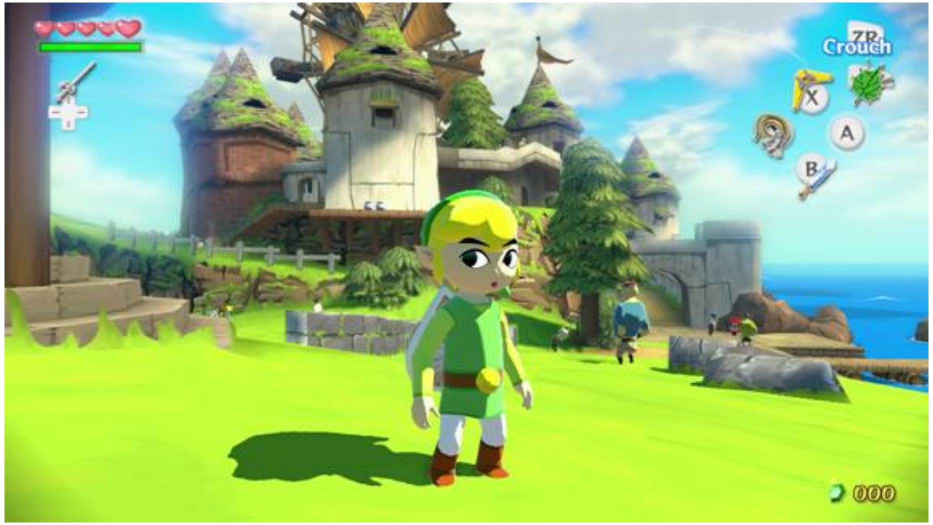
Cel Shading Examples

Wikipedia example



https://en.wikipedia.org/wiki/Cel_shading

The Legend of Zelda: The Wind Waker HD



http://www.rpgfan.com/previews/The_Legend_of_Zelda_The_Wind_Waker_HD/index.html

Cel Shading Algorithm

Part 1: Rendering the object outline (CPU code)

- For each triangle mesh object in the scene:
    - Create a copy of the object’s vertex buffer
    - For each triangle in the current mesh object:
        - Translate each vertex of the triangle outward along the direction of the vertex’s corresponding normal vector
        - Swap the order of the first two vertices in the triangle (changes the cyclic order of the triangle causing OpenGL to render the back half of the object)
    - Render the this object copy in black using OpenGL’s glDrawArrays function, with the first argument as GL_TRIANGLES (renders the back half of the slightly larger object copy)
- After each larger object copy is rendered, the corresponding object is rendered on top of the object copy everywhere except the boundary of the object copy, resulting in the black outline

Cel Shading Algorithm

Part 2: Rendering the discretized color gradient of the object (GLSL fragment shader code)

- Given: Some color generated by the Phong lighting model in the fragment shader at some point on the object, and the number of color “bins” used for cel shading (fewer bins results in more dramatic color gradients (difference in brightness between color and next brightest color increases)).
- Convert the RGB (Phong lighting) color value to the equivalent HSL (hue, saturation, and lightness) color value
- Create a new variable cel_lightness that is incremented from 0 by (1 / num_color_bins) until cel_lightness >= lightness from the RGB to HSL conversion
- Using the hue and saturation values from the RGB to HSL conversion and the new cel_lightness value, convert this HSL color back to RGB as cel_color
- Assign gl_fragColor = cel_color

Cel Shading Implementation Details and Results

- For the scenes we used, there can be an arbitrary number of lights, each of arbitrary colors, and every object can have ambient, diffuse, and specular colors that are different from one-another, so a cel-shading method that respected the variabilities of color for both lights and objects was developed
  - Cel shading done in simpler scene environments uses the monochromatic nature of objects to cleanly display discretized color gradient
- Monochromatic, dull objects tend to have the least dramatic color gradients and thus look the “least” cel shaded, while shiny, multichromatic objects tend to look the “most” cel shaded, with dramatic color gradients
- Demo outputs on next slides both used 6 as the number of color “bins” used

Cel Shading Results

Bunny2 scene:

light 5 5 5 , 0 0.6 0.8 , 0.025

objects:

bunny bunny.obj

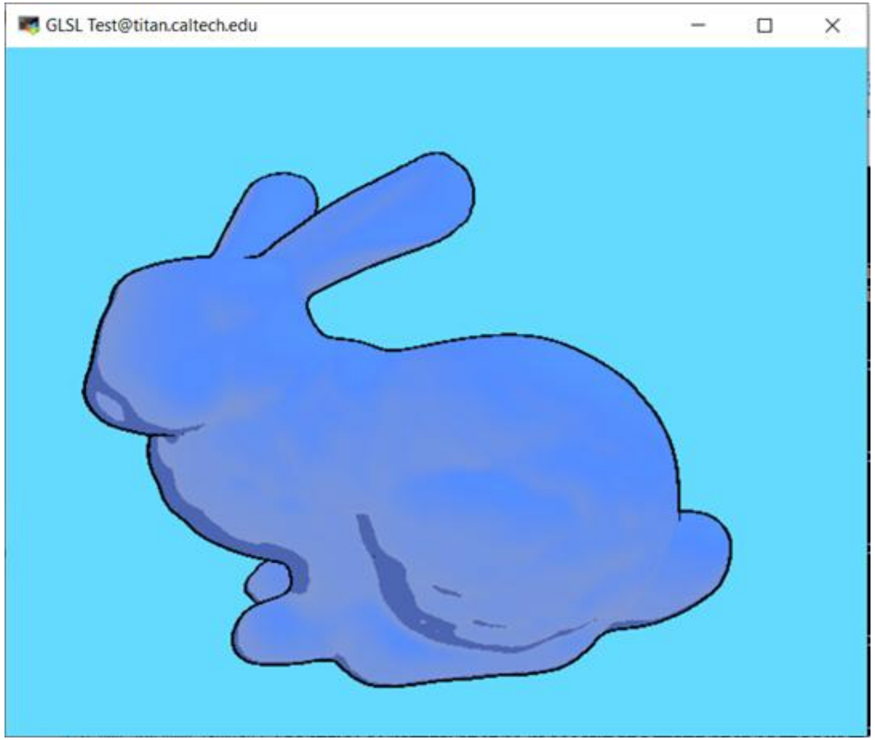
bunny

ambient 0.3 0.4 0.7

diffuse 0.3 0.4 0.7

specular 0.3 0.4 0.7

shininess 0.3



Cel Shading Results (Continued)

light 5 5 5 , 0 0.6 0.8 , 0

light -5 5 -5 , 0 0.1 0.2 , 0

objects:

armadillo armadillo.obj

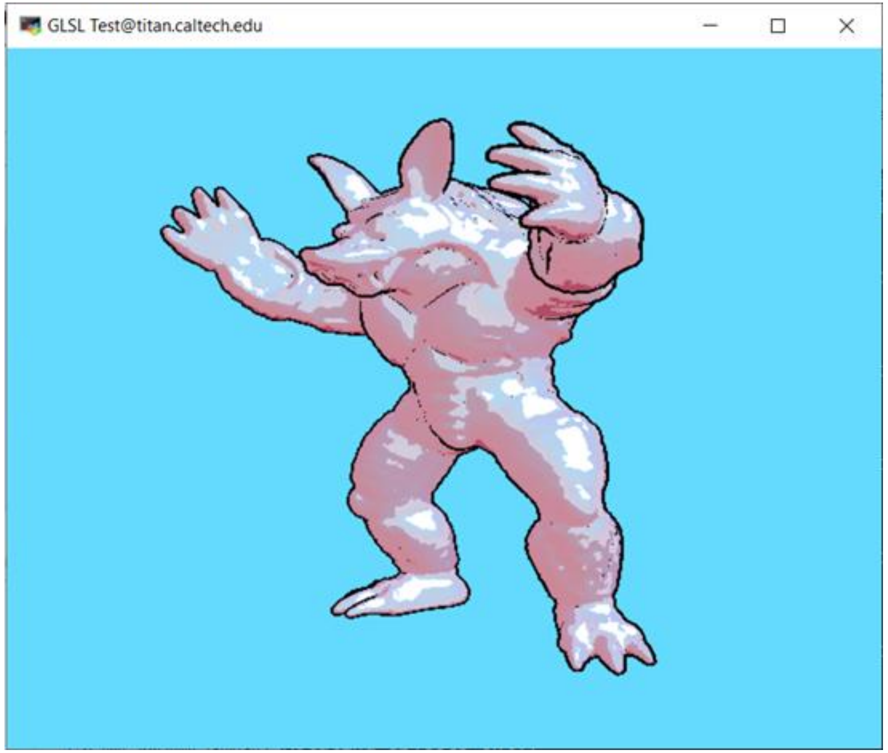
armadillo

ambient 0.7 0.3 0.4

diffuse 0.7 0.3 0.4

specular 0.3 0.7 0.4

shininess 2



Introduction: Water Wave Simulation

- Water waves are simulated using the wave equation: $$\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = \frac{1}{v^2}\frac{\partial^2 z}{\partial t^2}$$

- Much as finer and finer discretization of general functions allows closer and closer approximations of derivatives, we can get a good approximation of water via discretizing water into height values at many positions on a grid
- This is known as the height-field approach
- Alternatively, we can take the wave-particle approach, described in a 2007 SIGGRAPH paper, which uses the fact that water waves are almost always just superpositions of various frequencies of sin and cos waves

Water Wave Simulation Algorithm

- Initialize wave particles by picking a l_i (wavelength): $W_i(u) = \frac{1}{2}\left(\cos\left(\frac{2\pi u}{l_i}\right)+1\right)\Pi\left(\frac{u}{l_i}\right)$
- At each time step:
    - Update wave particles via the wave equation
    - Compute forces on objects in the water from each wave particle
    - Update each object’s position based on forces
    - GenerateWaveParticles ( ) RenderHeightFields ( )

Future Improvements to the Rendering Engine

- GPU ray tracing:
    - Debug the memory issue (or use a later version of CUDA that supports recursion?)
    - Store the KD-tree in the texture cache to improve memory lookups
    - Instead of having each thread process a pixel, shard the tree among the threads and pass rays around
- Cel shading:
    - Apply the monochromatic cel shading model to each object’s ambient, diffuse, and specular colors separately, then mix the colors together at the end to observe possible aesthetic differences from current model
- Water waves:
    - Combine with cel shaded render to create animated cel shaded water waves

References

References for Ray tracing:

Bala, K., Marschner, S. (2015). *Ray Tracing (Intersection)* [PDF document]. Retrieved from

https://www.cs.cornell.edu/courses/cs4620/2015fa/lectures/06rtintersectWeb.pdf

Shih, Min et al. "Real-Time Ray Tracing with CUDA." doi: 10.1007/978-3-642-03095-6_32.

Popov, Stephan et al. "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing."

doi: 10.1111/j.1467-8659.2007.01064.x.

References for Cel-shading:

Hutchins, Adam, and Kim, Sean. "Advanced Real-Time Cel Shading Techniques in OpenGL."

https://www.niwa.nu/2013/05/math-behind-colorspace-conversions-rgb-hsl/

https://en.wikipedia.org/wiki/Cel_shading

References for Water Surface Waves Simulation:

Yuksel, Cem et al. "Wave Particles." doi: 10.1145/1275808.1276501

Jeschke, Stefan. "Water Surface Wavelets." doi: 10.1145/3197517.3201336