CIS 441/541: Introduction to Computer Graphics
Lecture 17: rotations, ray tracing, parallel rendering
Announcements

- OH next week:
  - normal schedule
    - Tues 10-12
    - Weds 2-4
- Do we need to form special interest groups?
  - Camera manipulation
  - More?
The rest of this quarter

- **May 17th**: Lecture 13: geometry creation
- **May 22nd**: Lecture 14: buffers in GL
- **May 24th**: Lecture 15: transparency
- **May 29th**: Lecture 16: terrain rendering
- **May 31st**: Lecture 17: camera rotations, parallel rendering, ray tracing
- **June 5th**: Exam (worth 25% of your grade)
- **June 7th**: Lecture 18: Research with GPUs
The rest of this quarter

- May 17th / Lecture 13: geometry creation
- May 22nd / Lecture 14: buffers in GL
- May 24th / Lecture 15: transparency
- May 29th / Lecture 16: terrain rendering
- May 31st / Lecture 17: camera rotations, parallel rendering, ray tracing
- June 5th: Exam (worth 25% of your grade)
- June 7th / Lecture 18: Research with GPUs
Collision Detection

- Collision detection: as objects in the scene move, figure out when they collide and perform appropriate action (typically bouncing)

- Game setting: 30 FPS, meaning 0.033s to figure out what to render and render it.
  - Use spatial structures to accelerate searching
Collision Detection

- Two flavors:
  - A priori
    - before the collision occurs
    - calculate the trajectory of each object and put in collision events right before they occur
  - A posteriori
    - after the collision occurs
    - with each advance, see if anything has hit or gotten close

- Both use spatial search structures (octree, k-d tree to identify collisions)
Improved rotations

- **Project 2A/2B:**
  - Click here ➔ Medium rotations around combination of up & "up cross view" for as long as you hold button down.
  - Click here ➔ Small rotations around up axis for as long as you hold button down.
  - Click here ➔ Large rotations around "up cross view" for as long as you hold button down.
  - Click here ➔ Large rotations around up axis for as long as you hold button down.
Improved rotations: trackball

Only rotates while trackball is spun
Improved rotations: trackball

- Idea: approximate trackball interface with traditional mouse interface
- Camera movement occurs when the mouse is moving
- The camera does not move when the button is clicked, but the mouse does not move
Improved rotations: trackball

- Imagine your scene is contained within a sphere.
- When you push the mouse button, the cursor is over a pixel and the ray corresponding to that pixel intersects the sphere.
- Idea: every subsequent mouse movement (while the button is pushed) should rotate the intersection of the sphere so that it is still under the cursor.
Improved rotations: trackball

Click here

Move mouse here

Then sphere should move too
How to do the rotations?

- **Best way:** use quaternions
  - Number system that extends complex numbers
  - Applies to mechanics to 3D space
  - Would be a very long lecture!

- **Simple way:**
  - Take “dx” and “dy” in pixels, and then do
    - RotateAroundUp(dy*factor);
    - RotateAroundUpCrossView(dx*factor);
  - Factors vary based on level of zoom
  - Can create weird effects based on order of rotations
    - Users rarely notice in practice
Order matters for transparent geometry

- \((0,255,0,192)\) in front of \((255,255,0,192)\) = \((64,255,0)\)
- \((255,255,0,192)\) in front of \((0,255,0,192)\) = \((192,255,0)\)

Game plan:

- Sort geometry in back-to-front order, then render back-to-front.
  - Some optimizations on re-using sorting

Important GL calls

- `glEnable(GL_BLEND)`
- `glDepthMask(GL_FALSE)`
- `glBlendFunc`
- `glColor4ub(0,255,0,192)`
Introduction to Ray Tracing

Dr. Xiaoyu Zhang
Cal State U., San Marcos
Classifying Rendering Algorithms

- One way to classify rendering algorithms is according to the type of light interactions they capture.

- For example: The OpenGL lighting model captures:
  - Direct light to surface to eye light transport
  - Diffuse and rough specular surface reflectance
  - It actually doesn’t do light to surface transport correctly, because it doesn’t do shadows

- We would like a way of classifying interactions: *light paths*
Classifying Light Paths

- Classify light paths according to where they come from, where they go to, and what they do along the way
- Assume only two types of surface interactions:
  - Pure diffuse, D
  - Pure specular, S
- Assume all paths of interest:
  - Start at a light source, L
  - End at the eye, E
- Use regular expressions on the letters D, S, L and E to describe light paths
  - Valid paths are L(D|S)*E
Simple Light Path Examples

- **LE**
  - The light goes straight from the source to the viewer

- **LDE**
  - The light goes from the light to a diffuse surface that the viewer can see

- **LSE**
  - The light is reflected off a mirror into the viewer’s eyes

- **L(S|D)E**
  - The light is reflected off either a diffuse surface or a specular surface toward the viewer

- **Which do OpenGL (approximately) support?**
More Complex Light Paths

- Find the following:
  - LE
  - LDE
  - LSE
  - LDDE
  - LDSE
  - LSDE

Radiosity Cornell box, due to Henrik wann Jensen,
http://www.gk.dtu.dk/~hwj, rendered with ray tracer
More Complex Light Paths
The OpenGL Model

- The “standard” graphics lighting model captures only $L(D|S)E$
- It is missing:
  - Light taking more than one diffuse bounce: $LD^E$
    - Should produce an effect called color bleeding, among other things
  - Approximated, grossly, by ambient light
  - Light refracted through curved glass
    - Consider the refraction as a “mirror” bounce: $LDSE$
  - Light bouncing off a mirror to illuminate a diffuse surface: $LS+D+E$
  - Many others
  - Not sufficient for photo-realistic rendering
Raytraced Images

PCKTWTCH by Kevin Odhner, POV-Ray
Graphics Pipeline Review

- Properties of the Graphics Pipeline
  - Primitives are transformed and projected (not depending on display resolution)
  - Primitives are processed one at a time
  - Forward-mapping from geometrical space to image space

"Forward-Mapping" approach to Computer Graphics
Alternative Approaches: Ray CASTING (not Ray TRACING)

Ray-casting searches along lines of sight, or rays, to determine the primitive that is visible along it.

Properties of ray-casting:
- Go through all primitives at each pixel
- Image space sample first
- Analytic processing afterwards

"Inverse-Mapping" approach
Ray Casting Overview

- For every pixel shoot a ray from the eye through the pixel.
- For every object in the scene
  - Find the point of intersection with the ray closest to (and in front of) the eye
  - Compute normal at point of intersection
- Compute color for pixel based on point and normal at intersection closest to the eye (e.g. by Phong illumination model).
Ray Casting

- **Ray Cast (Point R, Ray D)** {
  - foreach object in the scene
    - find minimum $t > 0$ such that $R + t \ D$ hits object
  - if (object hit)
    - return object
  - else return background object
}
Raytracing

- Cast rays from the eye point the same way as ray casting
  - Builds the image pixel by pixel, one at a time

- Cast additional rays from the hit point to determine the pixel color
  - Shoot rays toward each light. If they hit something, then the object is shadowed from that light, otherwise use “standard” model for the light
  - Reflection rays for mirror surfaces, to see what should be reflected in the mirror
  - Refraction rays to see what can be seen through transparent objects
  - Sum all the contributions to get the pixel color
Raytracing

- Shadow rays
- Reflection ray
- Refracted ray
Recursive Ray Tracing

- When a reflected or refracted ray hits a surface, repeat the whole process from that point
  - Send out more shadow rays
  - Send out new reflected ray (if required)
  - Send out a new refracted ray (if required)
  - Generally, reduce the weight of each additional ray when computing the contributions to surface color
  - Stop when the contribution from a ray is too small to notice or maximum recursion level has been reached
Raytracing Implementation

- Raytracing breaks down into two tasks:
  - Constructing the rays to cast
  - Intersecting rays with geometry
- The former problem is simple vector arithmetic
- Intersection is essentially root finding (as we will see)
  - Any root finding technique can be applied
- Intersection calculation can be done in world coordinates or model coordinates
Constructing Rays

- Define rays by an initial point and a direction: \( x(t) = x_0 + td \)
- Eye rays: Rays from the eye through a pixel
  - Construct using the eye location and the pixel’s location on the image plane. \( x_0 = \text{eye} \)
- Shadow rays: Rays from a point on a surface to the light.
  - \( x_0 = \) point on surface
- Reflection rays: Rays from a point on a surface in the reflection direction
  - Construct using laws of reflection. \( x_0 = \) surface point
- Transmitted rays: Rays from a point on a transparent surface through the surface
  - Construct using laws of refraction. \( x_0 = \) surface point
From Pixels to Rays

\[ \vec{u} = \frac{\text{look} \times \text{up}}{|\text{look} \times \text{up}|} \]

\[ \vec{v} = \frac{\text{look} \times \vec{u}}{|\text{look} \times \vec{u}|} \]

\[ \Delta\vec{x} = \frac{2 \tan(f o_x/2)}{W} \vec{u} \]

\[ \Delta\vec{y} = \frac{2 \tan(f o_y/2)}{H} \vec{v} \]

\[ \vec{d}(i, j) = \frac{\text{look}}{|\text{look}|} + \frac{(2i + 1 - W)}{2} \Delta\vec{x} + \frac{(2j + 1 - H)}{2} \Delta\vec{y} \]
Ray Tracing Illumination

Recursive

\[ I(E,V) = I_{\text{direct}} + I_{\text{reflected}} + I_{\text{transmitted}} \]

\[ I_{\text{reflected}} = k_r I(P,V_{\text{reflected}}) \]

\[ I_{\text{transmitted}} = k_t I(P,V_{\text{transmitted}}) \]

\[ I_{\text{direct}} = k_a I_{\text{ambient}} + I_{\text{light}} \left[ k_d (\hat{N} \cdot \hat{L}) + k_s \left( -\hat{V} \cdot \hat{R} \right)^{\text{shiny}} \right] \]

Check for shadowing (intersection with object along ray \((P,L)\))
The Ray Tree

N_i surface normal
R_i reflected ray
L_i shadow ray
T_i transmitted (refracted) ray

Psuedo-code
Reflection

- Reflection angle = view angle

\[ \vec{R} = \vec{V} - 2(\vec{V} \cdot \vec{N})\vec{N} \]
Reflection

- The maximum depth of the tree affects the handling of refraction
- If we send another reflected ray from here, when do we stop? 2 solutions (complementary)
  - Answer 1: Stop at a fixed depth.
  - Answer 2: Accumulate product of reflection coefficients and stop when this product is too small.

0 recursion 1 recursion 2 recursions
Reflection
**Refraction**

Snell’s Law \[
\frac{\sin \theta_i}{\sin \theta_t} = \frac{n_i}{n_r} = \eta_r
\]

Note that I is the negative of the incoming ray.
Pseudo Code for Ray Tracing

rgb lsou;       // intensity of light source
rgb back;      // background intensity
rgb ambi;      // ambient light intensity

Vector L       // vector pointing to light source
Vector N       // surface normal
Object objects [n] // list of n objects in scene
float Ks [n]   // specular reflectivity factor for each object
float Kr [n]   // refractivity index for each object
float Kd [n]   // diffuse reflectivity factor for each object
Ray r;

void raytrace() {
    for (each pixel P of projection viewport in raster order) {
        r = ray emanating from viewer through P
        int depth = 1;  // depth of ray tree consisting of multiple paths
        the pixel color at P = intensity(r, depth)
    }
}
rgb intensity (Ray r, int depth) {
    Ray flec, frac;
    rgb spec, refr, dull, intensity;

    if (depth >= 5) intensity = back;
    else {
        find the closest intersection of r with all objects in scene
        if (no intersection) {
            intensity = back;
        } else {
            Take closest intersection which is object[j]
            compute normal N at the intersection point
            if (Ks[j] >0) { // non-zero specular reflectivity
                compute reflection ray flec;
                refl = Ks[j]*intensity(flec, depth+1);
            } else refl =0;
            if (Kr[j]>0) { // non-zero refractivity
                compute refraction ray frac;
                refr = Kr[j]*intensity(frac, depth+1);
            } else refr =0;
            check for shadow;
            if (shadow) direct = Kd[j]*ambi
            else direct = Phong illumination computation;
            intensity = direct + refl +refr;
        }
    }
    return intensity; }

Raytraced Cornell Box

Which paths are missing?

Ray-traced Cornell box, due to Henrik Jensen,
http://www.gk.dtu.dk/~hwj
Paths in RayTracing

- Ray Tracing
  - Captures LDS*E paths: Start at the eye, any number of specular bounces before ending at a diffuse surface and going to the light

- Raytracing cannot do:
  - LS*D+E: Light bouncing off a shiny surface like a mirror and illuminating a diffuse surface
  - LD+E: Light bouncing off one diffuse surface to illuminate others

- Basic problem: The raytracer doesn’t know where to send rays out of the diffuse surface to capture the incoming light

- Also a problem for rough specular reflection
  - Fuzzy reflections in rough shiny objects

- Need other rendering algorithms that get more paths
A Better Rendered Cornell Box
Large Scale Visualization with Cluster Computing

Linux Cluster Institute Workshop
October 1, 2004

Kenneth Moreland
Sandia National Laboratories
The Graphics Pipeline

Points

Lines

Polygons

Rendering Hardware

Geometric Processing
  Translation
  Lighting
  Clipping

Rasterization
  Polygon Filling
  Interpolation
  Texture Application
  Hidden Surface Removal

Frame Buffer
  Display
Parallel Graphics Pipelines

- Rendering Hardware
  - Geometric Processing
    - Translation
    - Lighting
    - Clipping
  - Rasterization
    - Polygon Filling
    - Interpolation
    - Texture Application
    - Hidden Surface Removal

- Frame Buffer
  - Display

- Each loaded/calculated individually

- Points
- Lines
- Polygons

Points, Lines, and Polygons are loaded/calculated individually.
Parallel Graphics Pipelines
Sort Middle Parallel Rendering

Sorting Network
Sort First Parallel Rendering
Sort Last Parallel Rendering

Sorting Network
Parallel rendering: ideas

- **Sort first:**
  - Reduce the amount of geometry considered
    - Depth culling (on node)
    - Decimation

- **Sort last:**
  - Reduce the number of pixels considered
    - “Active pixels”
    - Communication grouping
The rest of this quarter

✓ May 17\textsuperscript{th} / Lecture 13: geometry creation
✓ May 22\textsuperscript{nd} / Lecture 14: buffers in GL
✓ May 24\textsuperscript{th} / Lecture 15: transparency
✓ May 29\textsuperscript{th} / Lecture 16: terrain rendering
✓ May 31\textsuperscript{st} / Lecture 17: camera rotations, parallel rendering, ray tracing

☐ June 5\textsuperscript{th}: Exam (worth 25\% of your grade)
☐ June 7\textsuperscript{th} / Lecture 18: Research with GPUs