CIS 441/541: Introduction to Computer Graphics
Lecture 21: tests, interactors, raytracing, collision detection, animation
Talk about the test
Schedule

- Nov 21: today
- Nov 23: Kyle and Abhishek
- Nov 28: Chad and Jacob
- Nov 30: Mass OH for Project G
  - Grad students attend, help with projects
- Dec 2: final topics
Project G

- 3 prompts posted
- Final ones coming
- Remember: you choose 5 of 6
- Each to take ~8 hours
- $5 \times 8 = 40$ hours
  - Approx time for self-defined final project
Primary purpose of Hank’s OH is now to help with self-defined projects

Today: 3-4

Tuesday: 5-7?
  - With a special focus on key-press events?

Next week: canceled
  - I am on airplanes at both OH times
  - Could hold makeup OH on Friday

OH this weekend?
Non-Hank OHs

- Dan’s OH: no longer held
- Do we need “special OH” for Project G projects?
Outline

- Terrain rendering
- Spatial search structures
- View Frustum Culling
- Collision Detection
- Ray-Tracing
Terrain rendering

Wikipedia:

- Terrain rendering covers a variety of methods of depicting real-world or imaginary world surfaces.
  - Most common terrain rendering is the depiction of Earth's surface. Hank disagrees when it comes to CG

- Used in various applications to give an observer a frame of reference
Terrain rendering structure

- **Actors:**
  - terrain database,
  - a central processing unit (CPU),
  - a dedicated graphics processing unit (GPU),
  - a display.

- **Software application is configured to start at initial location in the world space.**

- **The output of the application is screen space representation of the real world on a display.**
Terrain rendering details

- CPU identifies and loads terrain data corresponding to initial location from the terrain database.
- CPU applies the required transformations to build a mesh of points that can be rendered by the GPU.
- Data sent to GPU and GPU completes geometrical transformations, creating screen space objects (i.e., polygons) that create a picture closely resembling the location of the real world.
The main tension is between the # of processed polygons and the # of rendered polygons.

- A very detailed picture of the world might use billions of data points.

- Virtually all terrain rendering applications use level of detail to manage number of data points processed by CPU and GPU.

- There are several modern algorithms for terrain surfaces generating.
Example: ROAM

- **ROAM**: Real-time optimally adapting mesh.
  - Continuous level of detail algorithm that optimizes terrain meshes.
  - Premise: sometimes it is more effective to send a small amount of unneeded polygons to the GPU, rather than burden the CPU with LOD (Level of Detail) calculations.
  - Result: produce high quality displays while being able to maintain real-time frame rates.
  - ROAM provides control over scene quality versus performance in order to provide HQ scenes while retaining real-time frame rates on hardware.
Figure 1: Example of ROAM terrain.

Figure 2: Triangulation for example frame, with eye looking right.
ROAM internals

to the midpoint $v_c$ of its base edge $(v_0, v_1)$. The left child of $T$ is $T_0 = (v_c, v_0, v_1)$, while the right child of $T$ is $T_1 = (v_c, v_1, v_0)$. The rest of the triangle bintree is defined by recursively repeating this splitting process.

Figure 3: Levels 0–5 of a triangle bintree.

4.2 Dynamic Continuous Triangulations

Meshes in world space are formed by assigning world-space positions $w(v)$ to each bintree vertex. A set of bintree triangles forms a continuous mesh when any two triangles either overlap nowhere, at a common vertex, or at a common edge. We refer to such continuous meshes as bintree triangulations or simply triangulations. Figure 4 shows a typical neighborhood about a triangle $T$ within a triangulation. A triangle $T$ in a triangulation cannot be split immediately when its base neighbor $T_B$ is from a coarser level. To force $T$ to be split, $T_B$ must be forced to split first, which may require further splits in a recursive sequence. A case requiring a total of four splits is depicted in Figure 5. Such forced splits are needed for the optimization algorithm in Section 5.
Outline

- Terrain rendering
- **Spatial search structures**
- View Frustum Culling
- Collision Detection
- Ray-Tracing
Spatial Search Data Structures

- Organize geometry so you can quickly find geometry in a given region.

- Examples:
  - Octree
  - k-d tree
  - Binary space partitioning

- Usages:
  - Collision detection
  - Culling
  - Ray tracing (Fri)
Octrees

- Oct + tree = octree (one ‘t’)
- Tree data structure
  - Internal node: has eight children, corresponding to 8 octants
  - Leaf node: contains some number of points
k-d trees

- **k-d = k-dimensional tree**
  - We are interested in k=3

- **Node roles:**
  - Every leaf node is a point
  - Every internal node divides space into two parts using a plane. Also contains a point.
    - Each plane is axis-aligned, i.e., X=a, Y=b, or Z=c.
    - Alternate between axes as you descend the tree
k-d tree
k-d tree
General form of k-d trees, using arbitrary planes, not just $X=a$, $Y=b$, $Z=c$

Associated tree (BSP trees) can be used for spatial searches.
Outline

- Terrain rendering
- Spatial search structures
- View Frustum Culling
- Collision Detection
- Ray-Tracing
Viewing frustum culling: the process of removing objects that lie completely outside the viewing frustum from the rendering process.

Rendering these objects would be a waste of time since they are not directly visible.
Spatial search structures (octree, k-d, BSP) can dramatically accelerate view frustum culling.

- Need correct granularity though.

Speedups are very application-dependent

- Best case scenario: you are zoomed in on very complex scene
- Worst case scenario: you are zoomed out on simple scene
Outline

- Terrain rendering
- Spatial search structures
- View Frustum Culling
- Collision Detection
- Ray-Tracing
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)
Outline

- Terrain rendering
- Spatial search structures
- View Frustum Culling
- Collision Detection
- Ray-Tracing
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: $L D^* E$
    - 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: $L D S E$
  - Light bouncing off a mirror to illuminate a diffuse surface: $L S + D + E$
  - Many others
  - Not sufficient for photo-realistic rendering
Raytraced Images

PCKWTWTCH by Kevin Odhner, POV-Ray
The previous slides now look like amateur hour...
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
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: $\mathbf{x}(t) = \mathbf{x}_0 + td$
- 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.
  - $X_0 = \text{point on surface}$
- Reflection rays: Rays from a point on a surface in the reflection direction
  - Construct using laws of reflection. $X_0 = \text{surface point}$
- Transmitted rays: Rays from a point on a transparent surface through the surface
  - Construct using laws of refraction. $X_0 = \text{surface point}$
From Pixels to Rays

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

\[ \mathbf{v} = \frac{\text{look} \times \mathbf{r}}{|\text{look} \times \mathbf{r}|} \]

\[ \Delta x = \frac{2 \tan(\text{fov}_x / 2) \cdot \mathbf{r}}{W} \cdot \mathbf{u} \]

\[ \Delta y = \frac{2 \tan(\text{fov}_y / 2) \cdot \mathbf{r}}{H} \cdot \mathbf{v} \]

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

Recursive

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

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

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

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

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

Viewpoint
Reflection

- Reflection angle = view angle

\[ \overrightarrow{R} = \overrightarrow{V} - 2(\overrightarrow{V} \cdot \overrightarrow{N})\overrightarrow{N} \]
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.
Reflection
Refraction

Snell’s Law \( \frac{\sin \theta_t}{\sin \theta_i} = \frac{\eta_i}{\eta_t} = \eta_r \)

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