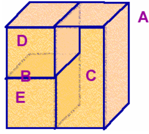
Hidden Surface Elimination
Raytracing



CS148: Intro to CG
Instructor: Dan Morris
TA: Sean Walker
August 2, 2005

Pre-lecture business

- Get going on pp4
- Submit exam questions by *Sunday*
- Remote folks: let us know if you have a different fax # than we used for the midterm or if you will not be taking the exam at 8:30PST

Review Quiz

- **What are the three components of the OpenGL lighting model?**
- **What's a *Bezier patch*?**
- **What piece of OpenGL state do we usually modify before drawing transparent objects?**
- **What is gluPickMatrix used for?**
- **How many points of extra credit does the highest-rated pp4 get?**
- **How many "extra" points do you get just for demo'ing your pp4 in class?**

Outline for today

- Hidden Surface Elimination
- Video Break
- Raytracing
- Terrain Generation

Hidden Surface Elimination (aka HSE/HSR)

- One definition of HSE is just making sure that objects don't get drawn on top of things that should be in front of them
- OpenGL uses the depth-buffer (Z-buffer) algorithm to solve this problem

What's another approach we could take to this problem?

This approach is called the "Heedless Painter's Algorithm".

Would this approach have any advantage over z-buffering?

So why do we use z-buffering instead?

Image-Space vs. Object-Space

- We call the heedless painter algorithm an "object-space" approach to HSE
- This approach is O(n log n) in the number of polygons in a scene

Why O(n log n)?

- Z-buffering an "image space" approach to HSE
- This approach is O(n•m) in the number of polygons in a scene (n) and the number of pixels in the framebuffer (m)

Why O(n•m)?

- What order are the worst-case *space* requirements for the two approaches?

A-buffering

- Like the z-buffer, but for each pixel, maintain a *sorted linked list* of all polygons that rasterized here
- When drawing each polygon **p**:

```
for each pixel this polygon affects
    insert p into this pixel's list
    if p is opaque and covers the entire pixel
        throw out everything farther away
    else
        store p's color and transparency
        store the amount of the pixel covered by p
```

- When every object has been drawn:

```
for each pixel
    blend together all the polygons on my list
```

A-buffering: pros and cons

- **What advantages does A-buffering offer?**
- **What disadvantages does A-buffering come with?**

HSR isn't just for correctness…

- These techniques basically avoided the "messed-up-TOBOR problem"
  - I.e. using *some* HSR technique is required to make a "correct" image
- In addition to hiding obscured surfaces, we'd also like to use HSR to save time
  - We'd like to avoid even *sending* hidden surfaces to GL.
  - If we have to deal with those surfaces, we'd like to pass them through as little of our pipeline as we can.

What's one HSR technique we've used in GL to *accelerate* rendering?

HSR is ***huge*** for games…

- Good HSR techniques are among the most important things that game hackers do at game companies
- Often we can throw out almost 100% of the polygons in our world before we ever start rendering
- This lets us use *much* more complex models, rendering techniques, physics, etc.

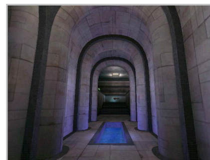
What surfaces are hidden?

- **What are three reasons that surfaces might be hidden in a game?**
  - Obscured: behind things, in another room
  - Too far away
  - Behind the camera / out of frustum

If I could throw away things outside a typical 45˚ FOV frustum, what fraction of my scene can I throw away?

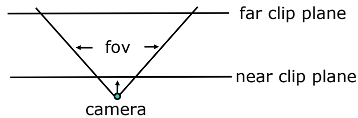
HSR affects game design…

- If these Quake maps have the same total number of polygons, which one is probably running faster and smoother?



View Frustum Culling

- View frustum culling: throwing away polygons that don't live in my view frustum
- For a typical perspective projection, what parameters define my view frustum?
  - I.e., if I wanted to test whether a point was in my view frustum, what would I need to know?



View Frustum Culling: Take One

- Assume we have a function:

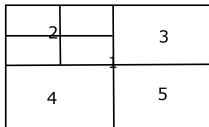
  bool polygonInFrustum(polygon p,
  frustum f);

for each polygon **p** in my scene
  if (polygonInFrustum(**p**,**f**))
    send this polygon to GL;

What's wrong with this approach? (or why isn't this any better…)

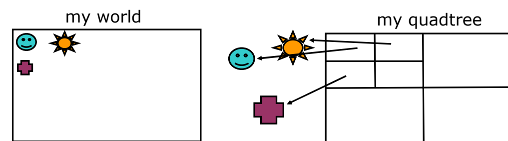
2D view culling: quadtrees

- A quadtree is a hierarchical data structure that divides 2D space up into boxes
- To build a quadtree for my 2D world, I would:
  - Create a root node that stores the four corners of my world
  - Divide my world into four boxes and create a new node for each box, storing his own corners
  - Store a pointer to each box in my root node
  - Divide each of those boxes into four boxes, and keep going until the boxes are "small enough"



2D view culling: quadtrees

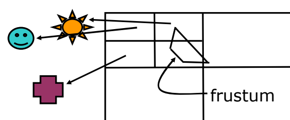
- Then I might add a pointer to every object in my world to the leaf (smallest) nodes of my quadtree
- Each object is pointed to by whatever leaf it "lives in"
- Empty nodes will be thrown away



View Culling with Quadtrees

- When it's time to render, I might do something like this:

```
list<node> nodes;
nodes.push_back(root node);
while(nodes.empty() == false)
    node n = nodes.pop_front();
    if n is a leaf
        if he's partially inside the frustum then render all his objects
        else continue
    else n must be an internal node
        if this node has no children then continue
        if this node is entirely outside the frustum then continue
        put all of this node's children on the nodes list
```



Quadtrees in practice

When will a quadtree not speed up my rendering at all?

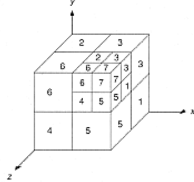
Describe the best-case speedup for a quadtree.

What objects might I want to render all the time, and never put in my quadtree?

Quadtrees are 2D data structures… when might they be useful in a 3D world?

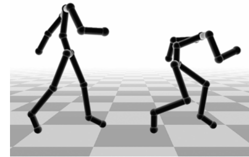
Octrees

- Octrees are the 3D cousin of quadtrees
- The world is recursively divided into eight boxes at each level of recursion, and culling proceeds just like it did for quadtrees
- Octrees provide very powerful view frustum culling



SIGGRAPH video break: Style Translation for Human Motion

- We saw in previous video breaks that animation was often the limiting factor in making 3D graphics realistic
- Goal of this paper: given manually modeled or captured "walking styles", apply those styles to new walking trajectories or new characters



Outline for today

- Hidden Surface Elimination
- Video Break
- Raytracing
- Terrain Generation

Interactivity vs. Realism

- So far, we've focused on interactive graphics
  - Everything has to render in about 50ms
  - Okay to sacrifice realism for speed
- Raytracing prioritizes realism over speed
  - Used for production videos, special effects, etc.

Raytracing in a Nutshell

- So far, we've done something like:

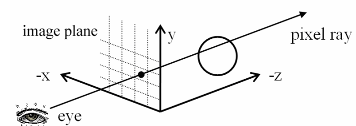
for each object
  send it through a lot of transformations to find out what pixels it ends up at

- Raytracing takes the opposite approach:

for each pixel
  figure out what objects should be visible at this pixel

Raytracing in a Nutshell

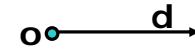
- Assume we know the camera position
- Assume we have chosen an "image plane" where our pixels are located
- We'll "trace a ray" from our camera through each pixel on the image plane, and see what it hits
- Really, we're modeling all the light that would have hit our eye coming through each of these pixels



Intersecting Rays

- Let's represent rays like this:

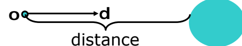
```
struct Ray {
    point origin;
    vector direction;
}
```



- Assume we have a function:

```
bool intersect(
        float& distance, ray R, Object O
);
```

  - Returns true if this ray hits this object
  - Sets "distance" to the distance from the ray origin at which the intersection occurs



Raytracing: Take One

```
for each pixel in the image plane {

    Object closestObject=NULL;
    float closestT=INFINITY;
    Ray r;
    r.origin = eye;
    r.direction = pixel – eye;

    for each object in our world {
            float t;
            if (intersect(t,r,object) == false) continue;
            if (t > closestT) continue;
            closestT = t;
            closestObject = object;
    }

    if (closestObject) set this pixel to closestObject.color;
}
```

What basic OpenGL feature does this approach lack?

Raytracing: Take Two

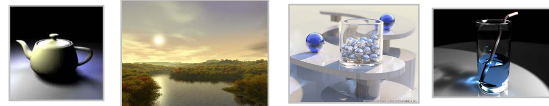
```
for each pixel in the image plane {

    …find the closest object…

    if (closestObject) {
            // Compute color just like we did in GL:
            I = I(amb) + I(diff) + I(spec);
    }
}
```

What other information do I need to take this approach?

What feature would OpenGL have to support to make it basically equivalent to this approach?

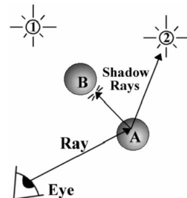
Something is still missing…

- What important features does "take two" lack?
  - Shadows
  - Reflection
  - Transparency
  - Refraction
- We can solve all these problems with one elegant solution: recursive ray tracing



Recursive Shadows

- Instead of just computing the lighting equation after we find the closest intersection, cast a new ray toward each light source
- If *this* ray hits any other objects before it hits the light source, our object is shadowed from this light

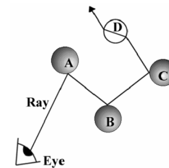


Reflection and Refraction

- If a ray hits a transparent, reflective, or refractive object, cast additional rays to find additional components of the pixel color…
- In other words, our lighting equation becomes:

$$I = I(amb) + I(diff) + I(spec) + k_{reflect} * I(refl) + k_{refract} * I(refr)$$

- $k_{reflect}$ is the "mirror-ness" of the object
- $K_{refract}$ is the "transparency" of the object
- I(refl) and I(refr) are the results of running our whole ray tracing operation on new rays



Computing reflected rays

- Given an incoming ray R and an intersection point, how would you compute the reflected ray direction?
  - **What information about the object do you need?**
  - **What do you know about the direction of the reflected ray?**
  - **Where in the CS148 lecture slides should I go to steal about this topic (i.e. where have we seen this before)?**

r = s′ – 2 (s′ • u**n**) u**n**

Computing refracted rays

- Given an incoming ray R and an intersection point, how would you compute the refracted ray direction?
- Snell's law: $\eta_i \sin \theta_i = \eta_t \sin \theta_t$
- Not going to go through the math here, but it's a good exercise in vector manipulation…
- For now, you should just see *what* data we need about the intersected object to give us a new ray

How would you generate an outgoing refracted ray for a translucent quad like this one?

Computing Intersection Points

- Ray representation re-visited:

r(t) = **o** + **d**t

- For a "primary ray" (a ray coming from the camera):
  - o = camera position
  - d = $(P_{pix} - P_{eye}) / | P_{pix} – P_{eye} |$
- An intersection function finds the t value – if any – where a ray hits an object
- In general, every type of object – polygonal meshes, spheres, voxel grids, etc. – will have its own intersection routine

Intersecting a Sphere

- Implicit function for a sphere:

| **p** – **c** | = R

c: center of sphere
p: any point on the sphere
R: radius

- To find where a ray hits a sphere, plug our ray into the implicit function for a sphere:
  - **p** = **o** + **d**t
  - | **o** + **d**t - **c** | = R
  - | **o** + **d**t - **c** $|^2$ = R^2
  - | (**o** - **c**) + **d**t $|^2$ = R^2
  - | **o** - **c** $|^2$ + 2 * | **o** - **c** | * **d**t + $|\mathbf{d}|^2 t^2$ = R^2

Intersecting a Sphere

- | **o** - **c** $|^2$ + 2 * | **o** - **c** | * **d**t + $|\mathbf{d}|^2 t^2$ = R^2
- This looks like a quadratic equation:

  $At^2 + Bt + C = 0$
  A = | **d** $|^2$
  B = 2 * | **o** - **c** | * **d**t
  C = | **o** - **c** $|^2$ - R^2

- Solutions are the intersections between our ray and our sphere:
  - $(-B \pm \sqrt{(B^2 – 4AC)}) / 2A$
- What does it mean for our raytracer if $(B^2 – 4AC)$ is:
  - **Negative**
  - **Positive**
  - **Zero**

Speeding up Raytracing

- Raytracing can take hours or days per frame
- Most of the time is spent in computing intersections

How could we speed up the intersection tests required for each ray?

- Raytracing also parallelizes really well…

How would you design a raytracer to run on many computers?

What if you had more computers than pixels?

Raytracing at home

- POV-Ray is a powerful, free raytracer used by everyone from designers who want pretty virtual prototypes to movie studios

http://www.povray.org

Outline for today

- Hidden Surface Elimination
- Video Break
- Raytracing
- Terrain Generation

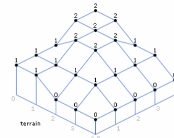
Terrain Generation
(many images taken from robot-frog.com)

- Many games use large outdoor scenes
- It can be very tedious to manually model hills, mountains, oceans, etc.
- Often games want the terrain to be a little different every time you play, or want "infinite terrain"
- Enter *Artifical Terrain Generation…*

Representing Terrain

- Most terrain can be represented as a 2D "height map"
- Each element in a 2D grid stores the height of the terrain at that position
- Can build OpenGL quads from this data

What types of terrain features can't be represented using height maps?

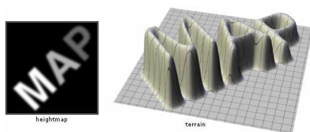


Terrain Generation: Take One

- Generate a 2D image in your favorite image editor
- Pixel intensities correspond to terrain height

What advantages does this have over modeling the terrain using 3D polygons?

What important terrain-generator properties does our "take one" system lack?

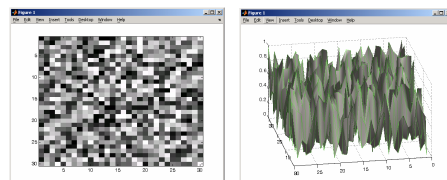


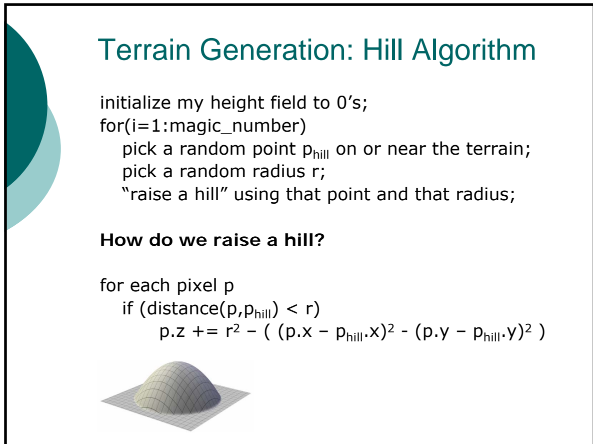
Terrain Generation: Take Two

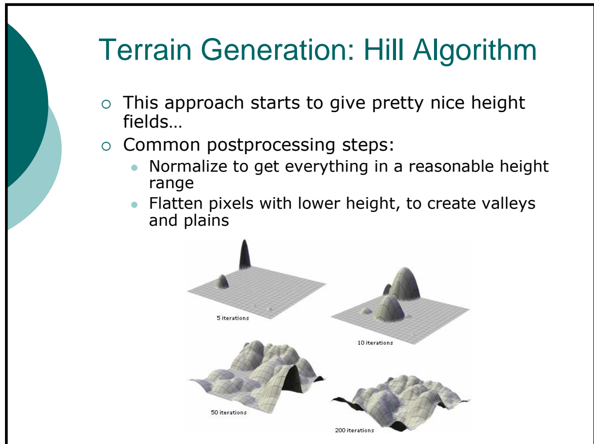
- Generate a random 2D height map

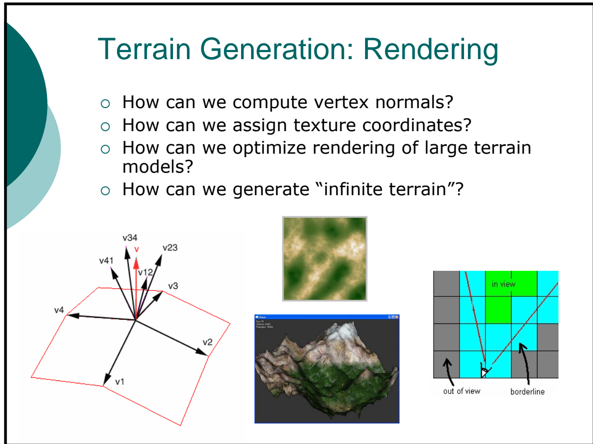
for all pixels in my image
 height = rand();

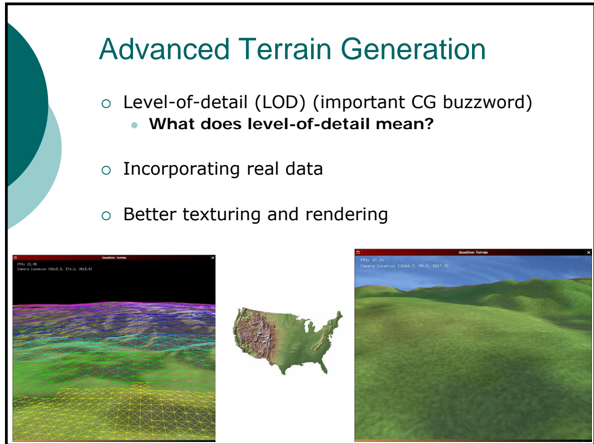
What's wrong with this approach?

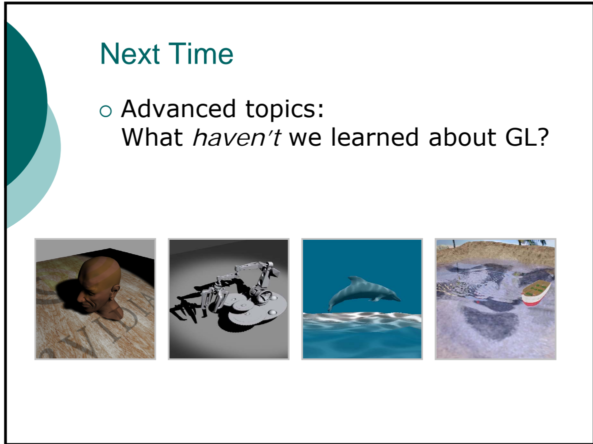



Terrain Generation: Hill Algorithm
initialize my height field to 0's;
for(i=1:magic_number)
pick a random point p_{hill} on or near the terrain;
pick a random radius r;
"raise a hill" using that point and that radius;
How do we raise a hill?
for each pixel p
if (distance(p,p_{hill}) < r)
$p.z \mathrel{+}= r^2 - ((p.x - p_{hill}.x)^2 - (p.y - p_{hill}.y)^2)$


Terrain Generation: Hill Algorithm
This approach starts to give pretty nice height fields…
Common postprocessing steps:
Normalize to get everything in a reasonable height range
Flatten pixels with lower height, to create valleys and plains
5 iterations
10 iterations
50 iterations
200 iterations


Terrain Generation: Rendering
How can we compute vertex normals?
How can we assign texture coordinates?
How can we optimize rendering of large terrain models?
How can we generate "infinite terrain"?
v34
v
v23
v41
v12
v3
v4
v2
v1
in view
out of view
borderline


Advanced Terrain Generation
Level-of-detail (LOD) (important CG buzzword)
What does level-of-detail mean?
Incorporating real data
Better texturing and rendering


Next Time
Advanced topics:
What haven't we learned about GL?