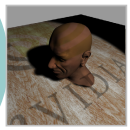
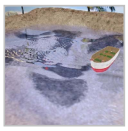
Collision Detection
Shadows
Programmable shaders



CS148: Intro to CG
Instructor: Dan Morris
TA: Sean Walker
August 4, 2005

Review Quiz

- What might I use a quadtree for?
- Name two approaches OpenGL uses for hidden surface removal
- What visual effects does recursive raytracing allow that non-recursive raytracing can't produce?
- What type of computation does a raytracer spend most of its time doing?
- What is Dan's recommended way of studying for CS148 exams?

Pre-lecture business

- Get going on pp4
- Submit exam questions by *Sunday*

Outline For Today

- Collision Detection
- Shadows
- Programmable Shaders

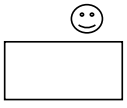
Collision Detection

- Many games and simulations spend most of their CPU time doing collision detection: *does object A intersect object B?*

When I might want to detect collisions in games?

- Let's take our first attempt at collision detection… assume we can test whether two triangles intersect:

```
bool findCollision() {
  for(every object o1)
    for(every triangle t1 in o1)
      for(every object o2 != o1)
        for(every triangle t2 in o2)
          if(t1 intersects t2) return true;
}
```



What's wrong with this approach?

Accelerated Collision Detection

What have we seen before that's like this problem?

- Octrees are very powerful for collision detection…

```
bool findCollision() {
  for(every object o1)
    for(every octree cell c1 that o1 lives in)
      for(every object o2 in c1)
        for(every triangle t1 in o1)
          for(every triangle t2 in o2)
            if(t1 intersects t2) return true;
}
```

What has this improved?

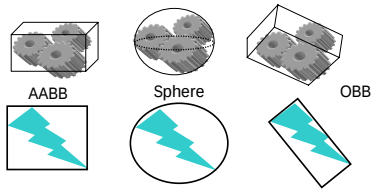
Bounding Volumes

- Very popular trick: store a *bounding volume* for each object:
  - BV: a large, simple shape that approximates the object's shape
  - Choose BV's for whom intersections are computationally easy
- Test collisions with bounding volumes before or instead of individual triangles
- Can combine with octrees to really speed up collision detection:

```
bool findCollision() {
  for(every object o1)
    for(every octree cell c1 that o1 lives in)
      for(every object o2 in c1)
        if(o1's BV intersects o2's BV)
          if(I don't care about precise collision detection)
            return true;
          for(every triangle t1 in o1)
            for(every triangle t2 in o2)
              if(t1 intersects t2) return true;
}
```

Bounding Volumes

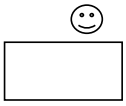
- Popular bounding volume shapes:
  - Axis-aligned bounding box
  - Oriented bounding box
  - Sphere



What are the tradeoffs between AABB and OBB?

Fixing other CD problems

- realtimerendering.com has a *great* table of free intersection code…
  - Shows you how to find intersections between rays/planes/sphere/cylinders/boxes/triangles/frustums /tetc.
- Let's revisit one of the other problems with our "take one" collision-detection approach:



If we represent these objects as a sphere and a box, how can we detect this collision even if they pass through each other in one frame?

Collision Detection Summary

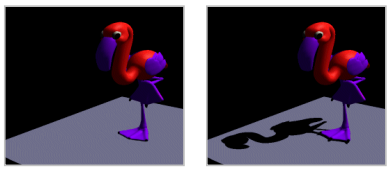
- Collision detection is a big CPU drain in games
- Hierarchical/spatial data structures are basically a requirement for CD in large worlds
- Figuring out how precise your CD needs to be can help you optimize
  - If my million-polygon spaceship can be a cone, I can do my CD much more quickly

Outline For Today

- Collision Detection
- Shadows
- Programmable Shaders

Why do we care about shadows?

- Realism
- Depth cues

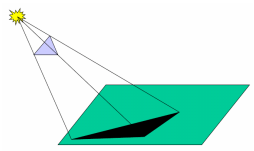


Shadowing algorithms

- Planar shadows
  - Very fast but can only cast shadows on planar objects
- Shadow maps
  - Flexible but don't produce great shadows
- Shadow volumes
  - Best-quality shadows but very expensive

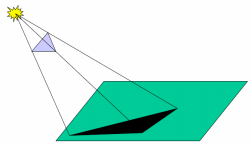
Planar Shadows

- Objects can cast shadows onto planes (and only planes)
- No occlusion testing; every shadowcaster casts a shadow on every shadowed plane
- **Sounds lame… why might this be useful?**



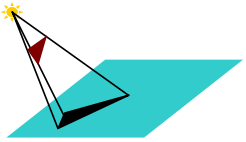
Planar Shadows

- The shadow is just rendered as another polygon, which happens to be a perspective-projected version of the shadow-caster
- **Where is the center of projection for this particular perspective projection?**



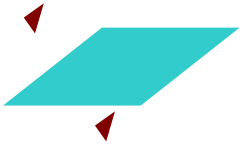
Planar Shadows: Implementation

- The big picture:
  - Render our world, including the floor and the shadowcaster as it would normally appear (no shadows)
  - Render the shadowcaster again in black, with a new modelview matrix that transforms it onto the floor
- Let's look at the special (but common) case where we want to cast shadows onto the plane y=0
  - Also assume the light is above the plane



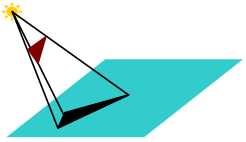
How do we build this transformation?

- I'll give you the first step: move the origin to the light source
  - translate(-lx,-ly,-lz);
- **What would happen if we rendered our polygon with only this translation?**



Getting some perspective…

- To get the perspective aspect of my transformation, I want z and x to get bigger as y gets more negative
- **What should my transformation matrix look like?**



w saves the day again…

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1/y & 0 & 0 \end{bmatrix}$$

- **What would happen if I translated, then applied this transformation, then rendered?**
- **What do I need to do to fix this?**

Almost there…

- Our steps so far:
  - Translate the origin to the light
  - Apply our perspective matrix
  - Translate the origin back
- **What should the y coordinate of my shadow's vertices be?**
- **How can we do that in OpenGL?**

And we have shadows…

- Projecting shadows onto y=0:
  - Translate the origin to the light
  - Apply our perspective matrix
  - Translate the origin back
  - Scale y to 0
  - Set the current color to black
  - Render our polygon
- **In our GL code, what order should the transformations appear in?**

Planar Shadows in GL [shadows.cpp]

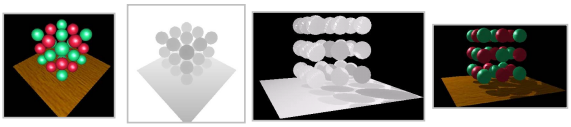
```
// This will be our projection matrix
float shadow_matrix[16];

for (i=0;i<15;i++) m[i]=0.0;
m[0] = m[5] =m[10] = 1.0;
m[7] = -1.0/lightpos.y;

glBegin(GL_POLYGON);
  // draw the polygon normally
glEnd();

glPushMatrix();
  glScalef(1,0,1);
  glTranslatef(lightpos.x,lightpos.y,lightpos.z);
  glMultMatrixf(shadow_matrix);
  glTranslate(-lightpos.x,-lightpos.y,-lightpos.z);
  glColor3fv(shadow_color);
  glBegin(GL_POLYGON);
    // draw the polygon again
  glEnd();
glPopMatrix();
```

Shadow maps

- Objects that are not *visible* to the light are shadowed
- **Does OpenGL give us a way to detect what objects are *visible* from a particular point in the scene?**

Shadow maps: the big picture

- Render the whole world from the position of the light source, but don't display it to the user
- The z-buffer is now a "shadow map"; copy it to a texture
- Render the world using the normal camera
- For each fragment at (x,y,z) that we want to render, transform it to the light's coordinate system (call it (x',y',z'))
- Compare z' to z = shadow_map[x',y']
- **What do I do if z' < z?**
- **What do I do if z' > z?**



Shadow maps in OpenGL (overview)

- The big steps we skipped past there were:
  - Transforming pixels into the light's coordinate frame
  - Coloring pixels depending on whether they're visible to the light
- If we can't do this in hardware, this isn't going to be helpful…
- Can use OpenGL's texture-generation functionality to generate texture coordinates for our objects on-the-fly
  - Texture coordinates can be set to the x,y,z offset of each vertex from the *camera*
  - But we want to know how far each vertex is from the *light*…
  - Can use the "texture transform matrix" to transform those values from eye space to light space
- Can use the "ARB_shadow" extension to automatically generate alpha or color values based on the results of the "shadow test"

Dan's lectures end here…

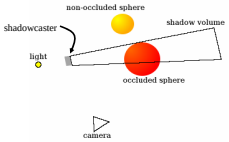
- This is as far as we got in class, so everything after this falls into the "not on the test" category.

Shadow maps: pros and cons

- Pros:
  - can shadow any object on any other object
  - uses lots of hardware acceleration so you do very little computation
- Cons:
  - requires one additional rendering pass for each light
  - can make ugly shadows, since the pixels in the two buffers don't necessarily line up exactly

State-of-the-art: shadow volumes

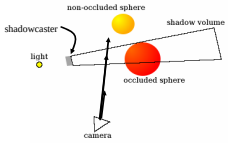
- Generate a whole polygon model that represents the volume of space shadowed by each object
- Test whether each object is inside (shadowed) or outside (lit) that volume



A raytracing perspective

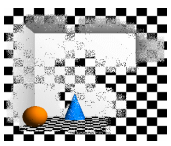
- Create an integer counter an initialize it to 0
- Cast a ray into the scene
- If we cross a front-face of a shadow volume, increment the counter
- If we cross a back-face of a shadow volume, decrement the counter
- **When I hit an object, how do I know whether it's in shadow?**

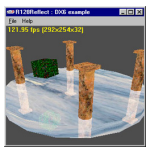
Q



Aisde: The Stencil Buffer

- The stencil buffer is a framebuffer-sized array of integers that you can manipulate
- Basically let's you allow rendering to an arbitrary portion of the screen… it's a framebuffer-sized *mask*
- Can increment, decrement, or set a pixel's value in the stencil buffer every time a pixel is rendered at that location
- Can throw out pixels if they fail some test, e.g. "this pixel is != 12 in the stencil buffer" or "this pixel > 33 in the stencil buffer"
  - The "stencil test" happens right after the depth test for each fragment
- Often used for special effects
  - Dissolving between two frames
  - Arbitrary viewports / reflections





Shadow Volumes in GL

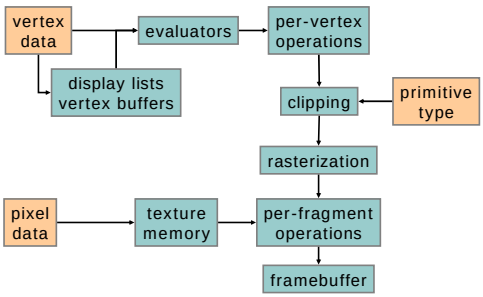
- The stencil buffer will be the counter we had in our raytracing example:
  - Render the front faces of the shadow volumes (not the objects) and increment the stencil buffer at each rendered fragment
  - Render the back faces of the shadow volumes (not the objects) and decrement the stencil buffer at each rendered fragment
- Turn on the stencil test so that pixels with a 0 in the stencil buffer will *not* be rendered
- Draw a big dark transparent quad over the entire screen; it will only be drawn on top of the *shadowed* pixels!

What were the expensive parts here?
When do I have to re-compute a lot of stuff?

Outline For Today

- Collision Detection
- Shadows
- Programmable Shaders

The OpenGL "fixed-function" pipeline



Shader evolution

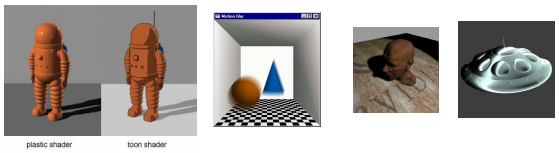
- In the beginning, there was transformation, lighting, texturing, and rasterization
  - Graphics cards had very dedicated circuits… a modelview-multiply circuit, a texture-fetch circuit, etc.
- As cards got faster, folks wanted more control over the pipeline…
  - Multiple textures, normal mapping, texture and color transformations, more complex materials, stencil tests, etc.
- At some point it became easier for hardware vendors to implement the pipeline with programmable mini-cpu's
- Then at some point they realized "hey, why not let the users program these mini-cpu's at run-time"

Enter shaders…

- Vertex shaders:
  - Small programs you can download to the graphics card
  - You can tell OpenGL: "*instead of* doing the regular T&L, run my program on every vertex"
  - Have nothing to do with shading
- Pixel shaders:
  - Small programs you can download to the graphics card
  - You can tell OpenGL: "*instead of* doing the regular fragment ops, run my program on every fragment"
- You now have a *programmable* OpenGL pipeline

Why do I want a programmable pipe?

- Effects we've already seen, e.g. bump mapping and environment mapping become much easier
- Complex new effects are possible



Shader programming languages

- Originally you had to write shaders in GPU assembly
  - **What does GPU stand for?**
- Even worse… different vendors had different assembly
  - So the good OpenGL folks defined a standard assembly language
  - Only so did the good DirectX folks
  - And they were both still assembly…
- Enter high-level shader languages…
  - You can now program your shaders using a language that looks just like C, and your driver will turn it into GPU assembly for you
  - The bad news is that there are still different languages for OpenGL and DirectX, and different languages for different vendors, but it's getting there…

What does a shader look like?

A sample of Nvidia's Cg shader language:

```
void simpleTransform(
    float4 objectPosition : POSITION,
    float4 color : COLOR,
    float4 decalCoord : TEXCOORD0,
    float4 lightMapCoord : TEXCOORD1,
    out float4 clipPosition : POSITION,
    out float4 oColor : COLOR,
    out float4 oDecalCoord : TEXCOORD0,
    out float4 oLightMapCoord : TEXCOORD1,
    uniform float brightness,
    uniform float4x4 modelViewProjection)
{
    clipPosition = mul(modelViewProjection, objectPosition);
    oColor = brightness * color;
    oDecalCoord = decalCoord;
    oLightMapCoord = lightMapCoord;
}
```

Is this a vertex shader or a pixel shader?

What does a shader look like?

A sample of Nvidia's Cg shader language:

```
float4 brightLightMapDecal(
    float4 color : COLOR,
    float4 decalCoord : TEXCOORD0,
    float4 lightMapCoord : TEXCOORD1,
    uniform sampler2D decal,
    uniform sampler2D lightMap) : COLOR
    {
        float4 d = tex2Dproj(decal, decalCoord);
        float4 lm = tex2Dproj(lightMap, lightMapCoord);
        return 2.0 * color * d * lm;
    }
```

Is this a vertex shader or a pixel shader?

GPGPU

- A hot area in graphics research right now: GPGPU == general-purpose GPU programming
- With programmable shaders, everyone has a limited but massively parallel computer on their desktop
- Harnessing this for physics computation in games and for scientific research is a hot topic (which sadly we don't have time to cover in CS148)

Check out http://www.gpgpu.org