Texture Mapping and Programmable Graphics Hardware

Rick Skarbez, Instructor
COMP 575
October 16, 2007

Announcements

- Programming Assignment 2 (3D graphics in OpenGL) is out
  - Due Thursday, October 25 by 11:59pm
- Programming Assignment 3 (Rasterization) is out
  - Due Thursday, November 1 by 11:59pm

Last Time

- Discussed programming assignments 2 & 3
- Presented the concept of Binary Space Partition (BSP) Trees
  - Described how they can be used to implement a painter's algorithm
- Began our discussion of texture mapping

Today

- More mapping
  - Finish up Texture Mapping
  - Bump Maps
  - Displacement Maps
- Discussion of programmable graphics hardwar
- Discussion of class project

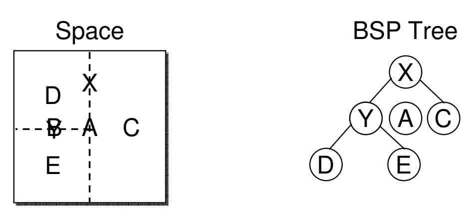
BSP Trees

Fuchs, Kedem, & Naylor; SIGGRAPH 1980

- Based on the concept of binary space partitioning
  - A plane divides space into two half-spaces; all objects can be classified as being on one side or the other
- A preprocessing step builds a BSP tree that <u>can be used for any viewpoint</u>
  - However, assumes that the geometry does not change

BSP Tree Illustration

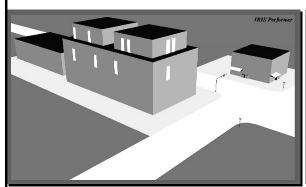
- Let's see an example (in 2D, not 3D)
  - Here, a line divides the plane into two half-planes



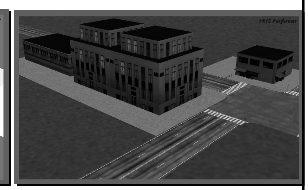
BSP Tree Review

- Use implicit planes to carve up space (and the geometry in it) into distinct subspaces
- One BSP tree can be used for *any* viewpoint
- Can be used to implement a painter's algorithm
  - Or to speed up a raytracer...
    - We'll be seeing this again
- Any questions?

Texturing Example



Before Texturing

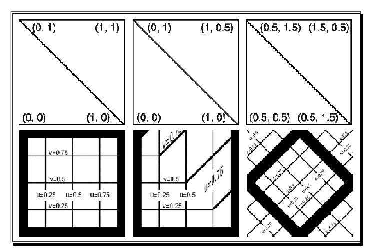


After Texturing

Texture Mapping

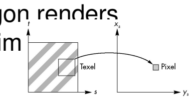
- Texture mapping allows us to render surfaces with very complex appearance
- How it works:
  - Store the appearance as a function or image
    - Take a picture
  - Map it onto a surface made up of simple polygons
    - Paste the picture on an object

Mapping Example

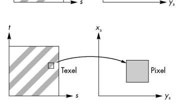


Sampling Issues

- So we can define the mapping, and it works fine
  - *As long as the size of the rendered image is approximately the same size as the texture source*
- What if the textured polygon renders much smaller in the final image than the original texture?



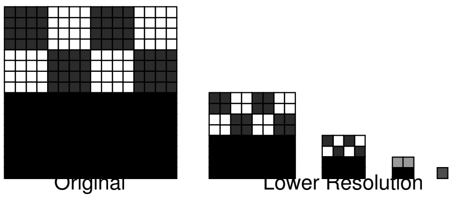
  - How about much bigger?



Mip-mapping to the Rescue

- Mip-mapping is a technique that creates multiple resolutions of an image
  - *i.e.* Takes a 512x512 image and filters it to create 256x256, 128x128, 64x64, ..., 1x1 versions of it
- Then, when you're looking up your texture coordinates, it uses the most appropriate mip-map level
  - Or, more likely, interpolates between the two closest

Mip-mapping Example



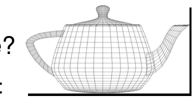
Original Texture

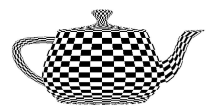
Lower Resolution Versions

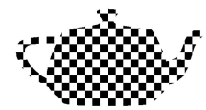
Original Images from David Luebke @ UVa

Assigning Texture Coordinates

- We generally want an even sharing of texels (pixels in the texture) across all triangles
- But what about this case?
  - Want to texture the teapot:



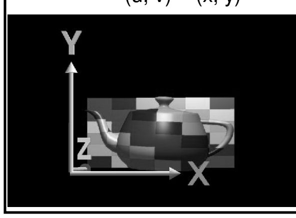


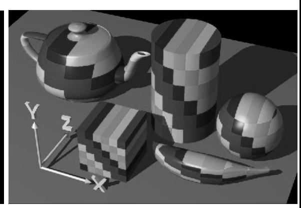


Do we want this? Or this?

Planar Mapping

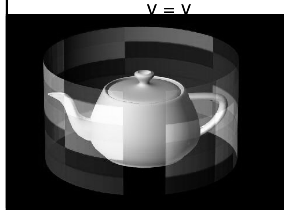
- Just use the texture to fill all of space
  - Same color for all z-values
  - (u, v) = (x, y)

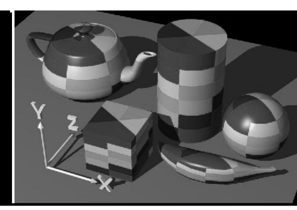




Cylindrical Mapping

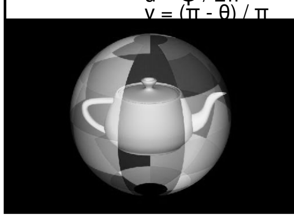
- "Wrap" the texture around your object
  - Like a coffee can
  - Same color for all pixels with the same angle
    - $u = \theta / 2\pi$
      $v = y$

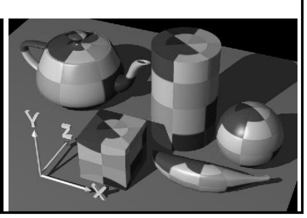




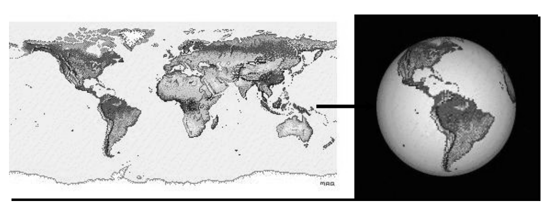
Spherical Mapping

- "Wrap" the texture around your object
  - Like a globe
  - Same color for all pixels with the same angle
    - $u = \varphi / 2\pi$
      $v = (\pi - \theta) / \pi$



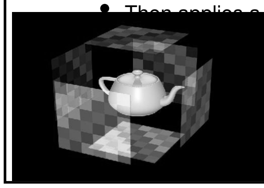


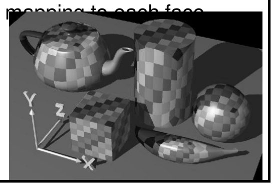
Spherical Mapping Example



Cube Mapping

- Not quite the same as the others
  - Uses multiple textures (6, to be specific)
- Maps each texture to one face of a cube surrounding the object to be textured
  - Then applies a planar mapping to each face

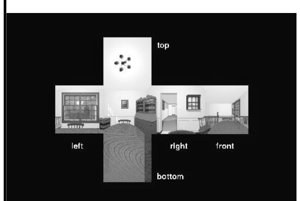




Environment Maps

- Cube mapping is commonly used to implement environment maps
- This allows us to “hack” reflection/refraction
  - Render the scene from the center of the cube in each face direction
  - Store each of these results into a texture
  - Then render the scene from the actual viewpoint, applying the environment textures

Environment Mapping Example

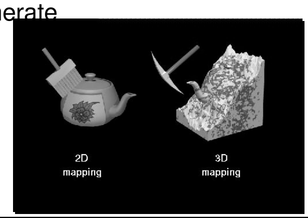


Solid Textures

- We’ve talked a lot about 2D (image) textures
  - Essentially taking a picture and pasting it on a surface
- No reason a texture HAS to be 2D, though
  - Can have 1D textures (not that interesting)
  - Can have 3D textures

3D Textures

- Actually, very easy to render with
  - Much simpler than 2D textures
- However, much more difficult to generate



Relevant OpenGL Functions

- glTexImage2D
- glEnable(GL_TEXTURE_2D)
- glTexParameter
- glBindTexture
- gluBuild2DMipmaps
- glTexEnv
- glHint(GL_PERSPECTIVE_CORRECTION, GL_NICEST)
- glTexCoord2df(s,t)

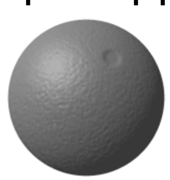
Texture Mapping Review

- Texture mapping is a relatively simple way to add a lot of visual complexity to a scene
  - Without increasing its geometric complexity
- Use mip-mapping to alleviate sampling problems
- There are infinitely many possible mappings
  - Usually want to use the most “similar” one
    - Texturing a plane? Use planar

Other Mapping Techniques

- So, now we know some things about texture mapping
  - Allows us to change the color of simple geometry
- But color isn’t the only property a point can have
  - Normals
    - Bump mapping
  - Location
    - Displacement Mapping

Bump Mapping



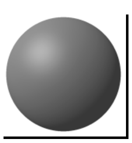
- How do we get this?
  - The underlying model is just a sphere

Bump Mapping

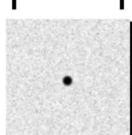
- Requires per-pixel (Phong) shading
- Just interpolating from the vertex normals gives a smooth-looking surface
- Bump mapping uses a “texture” to define how much to perturb the normal at that point
  - Results in a “bumpy” surface

Bump Mapping

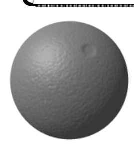
Note: Silhouette doesn’t change



Rendered Sphere



Bump Map

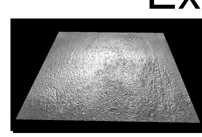


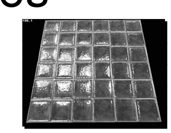
Bump Mapped Sphere

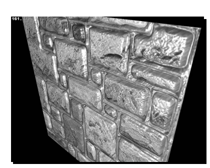
- At each point on the surface:
  - Do a look-up into the bump map “texture”
  - Perturb the normal slightly based on the “color”
    - Note that “colors” are actually just 3- or 4-vectors

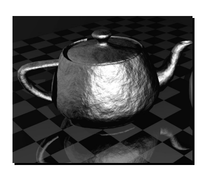
Images from Wikipedia

More Bump Mapping Examples









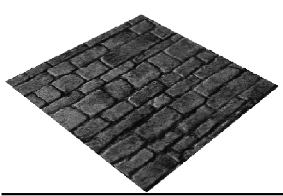
Displacement Mapping

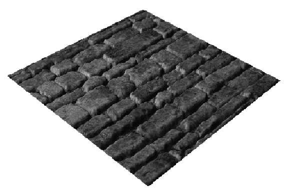
- Bump mapping adds realism, but it only changes the appearance of the object
- We can do one better, and actually change the geometry of the object
  - This is displacement mapping

Displacement Mapping

- Displacement mapping shifts all points on the surface in or out along their normal vectors
  - Assuming a displacement texture *d*,
    $\mathbf{p}' = \mathbf{p} + d(\mathbf{p}) * \mathbf{n}$
- Note that this actually changes the vertices, so it needs to happen in geometry processing

Displacement Mapping


Bump Mapping


Displacement Mapping

What does this buy us over bump mapping?

Movie Break! Red's Dream

Pixar, 1987

Available online:
http://www.metacafe.com/watch/47464/pixar_reds_dream/

Motivating Programmable Graphics Hardware

- Note that neither of these techniques can be implemented using the fixed-function pipeline that we've talked about so far
  - Bump mapping needs to delay lighting calculations until fragment processing
  - Displacement mapping needs to be able to do a lookup and edit vertices in the geometry step

Programmable Graphics Hardware

- Most recent graphics cards are programmable
  - Not quite like a CPU for various reasons
- On most hardware:
  - Replace the vertex processing stage with a programmable *vertex shader*
  - Replace the fragment processing stage with a programmable *fragment* (or *pixel*) *shader*
    - Some things are still fixed-function, like rasterization

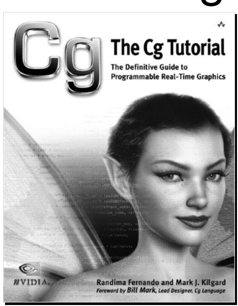
Shader Programs

- Vertex shader programs
  - Run on *each* vertex, independently
  - Output vertex properties (coordinates, texture coordinates, normal, color, etc.)
- Fragment shader programs
  - Run on *each* fragment, independently
    - Output the color of the fragment
    - Can also kill a fragment

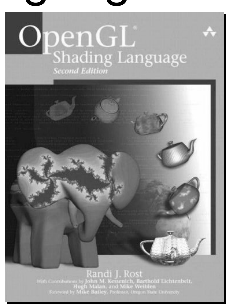
Programming Shaders

- Still a somewhat painful process
  - Somewhere between C and assembly in terms of difficulty
    - Cg is a bit lower level
    - GLSL is more like C
- Thankfully, most shader programs are short

Shading Languages



Cg



GLSL

Cg

- I can talk a little bit about Cg
- It's actually both a language and a runtime environment
  - Compiles your code down to machine language for your specific hardware
  - Can compile on the fly or ahead of time
    - Why choose one or the other?

Cg Vertex Program Example

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                            float3 normal   : NORMAL,

                        out float4 oPosition : POSITION,
                        out float3 objectPos : TEXCOORD0,
                        out float3 oNormal   : TEXCOORD1,

                    uniform float4x4 modelViewProj)
{
  oPosition = mul(modelViewProj, position);
  objectPos = position.xyz;
  oNormal = normal;
}
```

Cg Fragment Program Example

```
void C5E3f_basicLight(float4 position  : TEXCOORD0,
                      float3 normal    : TEXCOORD1,

                  out float4 color     : COLOR,

              uniform float3 globalAmbient,
              uniform float3 lightColor,
              uniform float3 lightPosition,
              uniform float3 eyePosition,
              uniform float3 Ke,
              uniform float3 Ka,
              uniform float3 Kd,
              uniform float3 Ks,
              uniform float  shininess)
{
  float3 P = position.xyz;
  float3 N = normalize(normal);

  // Compute the emissive term
  float3 emissive = Ke;

  // Compute the ambient term
  float3 ambient = Ka * globalAmbient;

  // Compute the diffuse term
  float3 L = normalize(lightPosition - P);
  float diffuseLight = max(dot(N, L), 0);
  float3 diffuse = Kd * lightColor * diffuseLight;

  // Compute the specular term
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(N, H), 0),
                            shininess);
  if (diffuseLight <= 0) specularLight = 0;
  float3 specular = Ks * lightColor * specularLight;

  color.xyz = emissive + ambient + diffuse + specular;
  color.w = 1;
}
```

Programmable Hardware Review

- Most modern graphics hardware is programmable
- Can write your own vertex processing and fragment processing
- There are several languages for shader programming, including Cg and GLSL
- Any questions?

Schedule for the Rest of the Semester

- Programming Assignment 2 due 10/25
- Programming Assignment 3 due 11/1
  - These already out
- Assignment 3 due 11/8
- Final Project Proposal due 11/8

Schedule for the Rest of the Semester

- Programming Assignment 4 due approx. 11/20
  - Raytracing
- Final Exam -- Friday 12/14 @ 4:00pm
- Final Project due approx. 12/6
  - Can be flexible with this

Final Project

- Pretty much open-ended
  - Can work on whatever you think is interesting
  - Should be roughly 1.5-2.5x a regular assignment
- Proposal due 11/8
  - You must meet with me before then to discuss your project
  - The "proposal" is a short (< 1 page) document that summarizes your project

Final Project "Topics"

- Make a game
  - Something more graphically advanced than assignments 1 or 2
- Implement some advanced OpenGL techniques
  - Shadows, environment mapping, etc.
- Implement something interesting with programmable shading
  - Displacement mapping, toon shading, etc.

Final Project "Topics"

- Add some advanced features to the raytracer
  - Depth-of-field, soft shadows, caustics, etc.
    - These will become clear later
- Implement a full rasterizer
  - Extend your rasterizer from assignment 3 to do lighting, texture mapping, etc.

Final Project "Topics"

- Implement some advanced UI
  - Use a webcam, joystick, Xbox controller, etc. to do something interesting
- Implement some functional/analytic graphics
  - Bezier curves, splines, etc.
  - Fractals
- Implement some image processing tools
  - *i.e.* Photoshop

Final Project "Topics"

- Generate some sufficiently advanced animation sequence
- Implement some high-dynamic range tone-mapping techniques
- If any of this (or anything else) interests you, and we haven't yet covered it in class, contact me and I'll point you to some info

Next Time

- Enjoy your fall break!
- When we come back, it's on to raytracing