Project 3 (textures)

Andrzej Szymczak

October 21, 2013

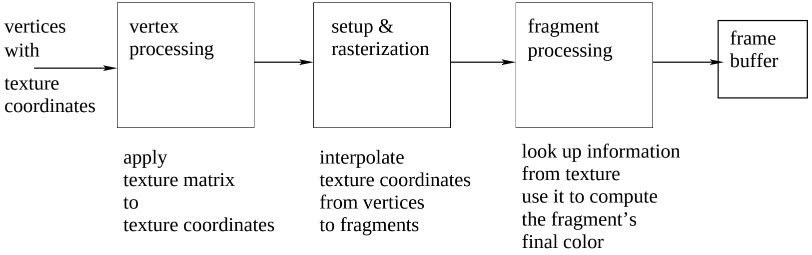
Goal

- Apply textures to 3D models
- Three sub-projects
  - 3D texture (simulated carving)
  - Environment mapping (mirror surface)
  - Tilable texture on a torus (you'll need to build the geometry)

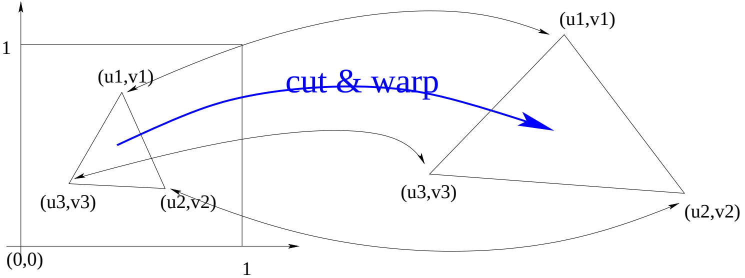


Texture mapping in graphics pipeline

- Specify *texture coordinates* for every vertex (as an attribute), or derive them from other attributes (location, normals etc)
  - Where should the color be looked up from?
- Functions to look up from texture provided in GLSL
  - Array textures become *samplers* in GLSL
  - Procedural textures can be implemented as functions that use a formula to compute a value based on texture coordinates



Texture mapping in graphics pipeline



texture
space

3D triangle
with texture coordinates
at the vertices

3D texture

- Input texture: $128 \times 128 \times 128$ 3D RGB image
- Raw format (sequence of RGB values for consecutive pixels/voxels on consecutive slices)
- Binary file, $3 * 128^3$ byte long

3D texture: carving

- Texture coordinates: Vertex coordinates scaled to $[0, 1]$ range
  - Use bounding box
  - Similar to the normalizing transformation in project 2, but scales to 0...1, not to -1...1

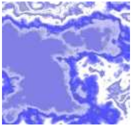
3D texture

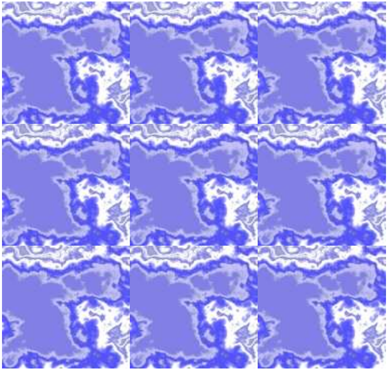
- Use bounding box info (from project 1)
- Transformation: first translate by $(-x_{min}, -y_{min}, -z_{min})$, then scale by $\frac{1}{\max(x_{max}-x_{min}, y_{max}-y_{min}, z_{max}-z_{min})}$
  - Easy to express algebraically without a matrix (add $(-x_{min}, -y_{min}, -z_{min})$ and scale the result)
- No need to use explicit attribute for texture coordinates (scaled locations are good texture coordinates!)
- Apply the transformation above on vertex processing stage and put the result in an output variable of type `vec3`
- Request perspectively correct (default) interpolation for that variable
- Use the interpolated value to look up color from texture in the fragment shader

3D texture

- Troubleshooting
  - Does the texture appear on the model? [OpenGL settings? TAPs?]
  - Isn't it too constant color? [Scaling by too much?]
  - Does the texture seem stretched on some parts of the model [2D texturing? too little scaling? wrong translation amount?]
  - Are there any seams along a planar cut through the model? [too little scaling? wrong translation amount?]

3D texture: origin of seams (repeat mode)

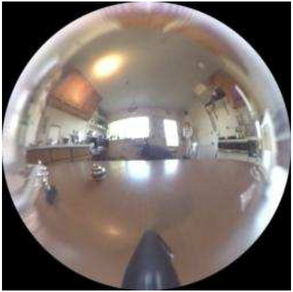




Example: volumetric textures

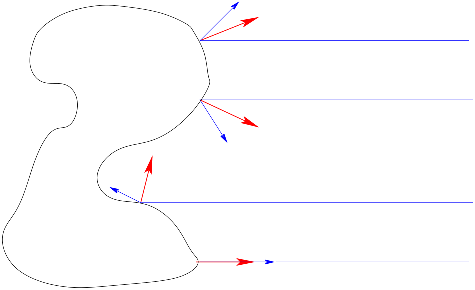
Spherical environment map

- A single photograph of a mirror sphere placed in the region of interest
- No need to stitch images together (one image is enough)
- Ideally, use a long [zoom] lens – you want to get as close to a parallel projection as you can
- Several environment maps are provided in the ppm format
- Code for reading the images (without any comment lines) is also provided



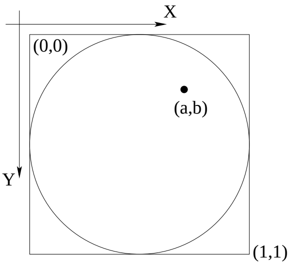
Mirror objects using the spherical environment map

- Pretend view is parallel when determining texture coordinates
- But use perspective projection to render the model
- Use a simple approximation that ignores multiple reflections
- Key observation: reflected ray direction determined by the normal



Mirror objects: math

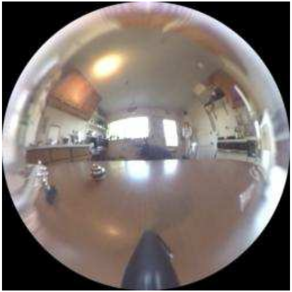
- Assume the projection is in the direction of z-axis
- Assume mirror sphere centered at the origin and of unit radius
- Normal to the mirror sphere under the point (a, b) (in texture space): $(2a - 1, 2b - 1, \sqrt{1 - (2a - 1)^2 - (2b - 1)^2})$
- ... therefore, point on the sphere with unit normal (n_x, n_y, n_z) is under the point $\left[\frac{1+n_x}{2}, \frac{1+n_y}{2}\right]$ in the texture space
- Texture coordinate for a vertex with unit normal (n_x, n_y, n_z): $\left[\frac{1+n_x}{2}, \frac{1+n_y}{2}\right]$; point on sphere under that point and the vertex have the same normals



Implementation

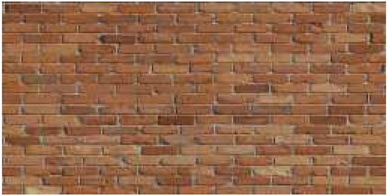
- Use normalized vertex normals as texture coordinates
- 3D texture coordinates, even though the texture is 2D
  - Easier to transform texture coordinates correctly
- Texture matrix
  - Rotate (apply the same matrix as you do for the model)
  - Translate by $(1, 1, 1)$
  - Scale uniformly by 0.5
- Troubleshooting
  - Use the synthetic hills probe to test if your reflection works (sky and ground reflections appear in the right places)
  - Use the provided sphere model (the output should be pretty much identical to the texture)
  - Make sure the reflections change correctly as the model is rotated
  - Black spots may be a sign of texture matrix issues or non-normalized normals
  - Problems with per-vertex normals that we missed in project 2 may show up!

Mirror objects: Feline in the kitchen

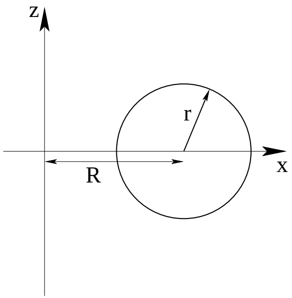


Tilable texture on a torus

- Glue horizontal edges to get a brick cylinder
- Glue the ends of the cylinder to get brick torus
- We'll use brick texture for grading
- All textures provided in ppm format
- No seams should be visible



Torus



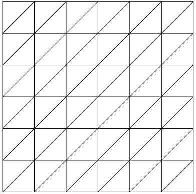
- Rotate circle as shown above around the z-axis
- Parametric equation of the circle
  $c(\psi) = (R + r\cos\psi, 0, r\sin\psi), \psi \in [0, 2\pi]$.
- Rotation matrix: $R_\phi = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \phi \in [0, 2\pi]$

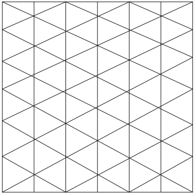
Torus

- Parametric equation of the torus

$$P(\psi,\phi) = R_\phi c(\psi) = ((R+r\cos\psi)\cos\phi, (R+r\cos\psi)\sin\phi, r\sin\psi).$$

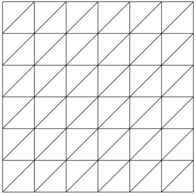
- This is a mapping of the square $[0,2\pi]\times[0,2\pi]$ onto the torus
- Triangulate the square and map triangles to 3D using P
- Strips easy to build (but not required)
- Use $(\psi/(2\pi), \phi/(2\pi))$ or $(\phi/(2\pi), \psi/(2\pi))$ as texture coordinates for vertex $P(\psi,\phi)$
- ... and $\pm\frac{\partial P}{\partial\phi}\times\frac{\partial P}{\partial\psi}$ as a normal

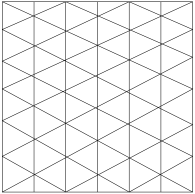




Torus

- P can be evaluated in the CPU code or in the vertex shader
- If in vertex shader, send a vertices of a triangulation of the parameter square and do this in the vertex shader:
  - apply P to transform vertices from parameter domain to 3D
  - apply the usual matrices as the shader provided in project 2 does to map to screen coordinates
  - also, ψ, ϕ can be used to generate texture coordinates, that can be sent out in an output variable (here, `vec2` type suffices)





Technicalities: creating texture objects

```
// global variables
RGBTexture2D *t2D = NULL;
RGBTexture3D *t3D = NULL;

// somewhere in code executed just once before rendering
  t2D = createRGBTexture2D("textures/mines.ppm",
                           0.5,0.25,0.25);
  t3D = createRGBTexture3D(128,128,128,
                           "textures/marble.rgb");
```

Technicalities: manipulating textures

```
t2D = createRGBTexture2D("textures/mines.ppm",
                         0.5,0.25,0.25);
t2D->linear();
t2D->clampToBorder();
t2D->attach(1);
t2D->on();

t3D = createRGBTexture3D(128,128,128,
                         "textures/3D/marble.rgb");
t3D->attach(2);
t3D->linear();
t3D->clampToEdge();
t3D->on();
```

Technicalities: GLSL

- Using the 3D texture in a shader
- `tcoord` and `fragcolor` are of type `vec3`
  - remember the return value of `texture` is of type `vec4`

```
layout (binding=2) uniform sampler3D tex;

// somewhere in the shader code
  fragcolor = texture(tex,tcoord).rgb;
```

Technicalities: GLSL

- Using the 2D texture in a shader
- `tcoord` is of type `vec2` and `fragcolor` is of type `vec3`
  - remember the return value of `texture` is of type `vec4`

```
layout (binding=1) uniform sampler2D tex;

// somewhere in the shader code
  fragcolor = texture(tex,tcoord).rgb;
```