Introduction to VGP353

- Agenda:
  - Course road-map
  - Render-to-texture techniques
    - Render to framebuffer, copy to texture
    - Framebuffer objects
  - Assign first programming assignment

© Copyright Ian D. Romanick 2007

Road-map

- Two new general OpenGL features:
  - Render to texture
    - Rendering to the framebuffer, then copy to a texture
    - Rendering directly to a texture via framebuffer objects
  - Stencil buffer
- Three general methods for generating shadows
  - Render planar shadows to a texture
  - Shadow maps
  - Shadow volumes.

© Copyright Ian D. Romanick 2007

Grading, etc.

- Assignments:
  - 5 programming assignments
    - You will have 2 weeks for most of them
  - 1 paper presentation
  - 1 term project
    - You will have 3 weeks for this
- Tests:
  - 4 *short* quizes
  - 1 long final :)

© Copyright Ian D. Romanick 2007

Rendering to a texture

- Several methods exist in OpenGL to render to a texture.
  - Render to the framebuffer, then copy the results to a texture.
  - Use the *new* framebuffer objects extension.
  - Render to a pixel buffer (pbuffer), then bind the pbuffer to a texture.
    - This method is platform dependent (i.e., is different on Linux, Windows, and Mac OS) and will *not* be covered in this course.

© Copyright Ian D. Romanick 2007

Why render to a texture?

- Many, many effects can be created by rendering to one or more textures, then using those textures to render the final scene.

© Copyright Ian D. Romanick 2007

Copy to texture

- Easiest and least efficient form of render-to-texture.
- Draw to the backbuffer, copy resulting image to texture with either `glCopyTexImage2D` or `glCopyTexSubImage2D`.
- *That's it.*

© Copyright Ian D. Romanick 2007

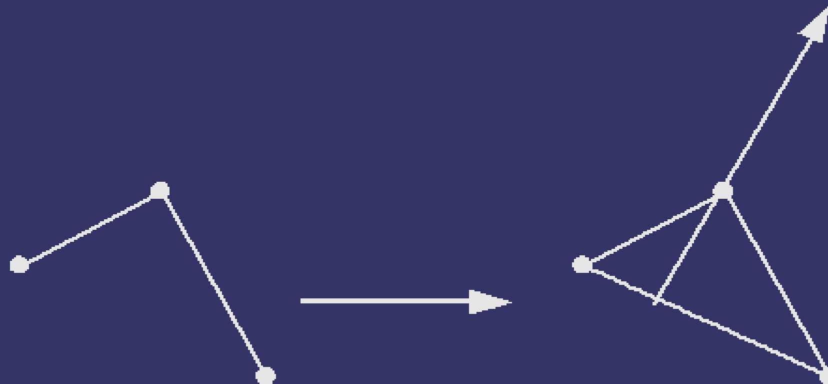
Problems with copy-to-texture

- Must perform extra copies.
- Must perform extra buffer clears.
- If the window is obscured or off the screen, the texture may be corrupted.
- The window must be at least as large as the desired texture.

© Copyright Ian D. Romanick 2007

Example: Normal Map Generation

- Given a height map texture, generate a normal map.
- The X component of the normal is the inverse of the slope of the line between the east and west neighboring texels.
  - Same for Y, but use the north and south neighbors.



© Copyright Ian D. Romanick 2007

Example: Normal Map Generation (cont.)

- *Really* easy to do in a fragment shader!
  1. Draw a single quad with texture coordinates ranging from 0 to 1 in both dimensions.
  2. Read the 4 texels around the current texel. Call them n, s, e, and w.
  3. Normal direction is:

$$d = vec3(e.x - w.x, s.y - n.y, 0.0)$$

$$d.z = 1.0 - \sqrt{(d \cdot d)}$$

$$d = normalize(d)$$

© Copyright Ian D. Romanick 2007

Example: Wave simulation

- If we have a height map that represents waves, we can simulate motion as a spring network.
  - Each wave is “pulled” up or down by the surrounding water.
- We need to track the wave position and velocity from time step to time step.
  - Store position in R, G, and B; velocity in A.
- Also need wave mass, spring constant, and time step size as uniforms.

© Copyright Ian D. Romanick 2007

Example: Wave simulation (cont.)

```
void main(void)
{
    vec4 me = texture2D(wave_state, gl_TexCoord[0].xy);
    vec2 f_vec = vec2(-4.0 * me.x, 0.5 - me.x);

    f_vec.x += texture2D(wave_state, north).r;
    f_vec.x += texture2D(wave_state, south).r;
    f_vec.x += texture2D(wave_state, east).r;
    f_vec.x += texture2D(wave_state, west).r;

    float F = dot(spring_constant, f_vec);
    float V = (mass * F) + (me.w - 0.5);
    float H = (time * V) + (me.x * damping);

    gl_FragColor = vec4(H, H, H, V + 0.5);
}
```


© Copyright Ian D. Romanick 2007

Example: Wave simulation (cont.)

- Add some damping and a force pulling the waves towards rest (i.e., 0.5) to stabilize the simulation.
- The resulting texture can be used as a gray-scale texture or to generate a normal map.
- Remember to adjust `time` to accurately measure frame time.
- Waves will eventually die.
  - Draw new waves into texture periodically.

© Copyright Ian D. Romanick 2007

Framebuffer Objects

- The framebuffer object (FBO) interface has a fairly steep learning curve.
  - We're just going to scratch the surface today, and we'll continue next week.
  - The ARB spent two years developing this interface.
  - It builds on the familiar texture interfaces, but is still *very* different.
- Now that I've stricken terror into your hearts...

© Copyright Ian D. Romanick 2007

Creating an FBO

- The first step is to create the FBO.
  - Use `glGenFramebuffersEXT` and `glBindFramebufferEXT`.
- Attach one or more renderable objects to it.
  - There are several functions available to do this. More on this later.
  - Conceptually, this is similar to attaching shader objects to a program object.
  - Example: Attach an RGBA texture to the FBO.

© Copyright Ian D. Romanick 2007

Using an FBO

- Once the FBO has all of its attachments:
  - Make sure the FBO is acceptable to the driver / hardware with `glCheckFramebufferStatusEXT`.
    - Some hardware can't handle some combinations of attachments.
    - Some combinations of attachments are just plain wrong (i.e., attaching a depth texture to a color attachment).
  - Bind the framebuffer with `glBindFramebufferEXT`.
  - Reset viewport and draw!

© Copyright Ian D. Romanick 2007

Using an FBO (cont.)

- When done rendering to FBO, bind the 0 object to resume rendering to window.
- To use textures that were rendered to, simply bind and use as usual.
  - You **cannot** use `GL_GENERATE_MIPMAPS` with FBO-rendered textures.
  - Instead, use new function `glGenerateMipmapEXT` to generate the mipmap stack on-demand.

© Copyright Ian D. Romanick 2007

Renderbuffers and textures

- Two broad types of objects can be attached to an FBO.
  - A texture. Most textures are both texturable and renderable.
  - A renderbuffer. Renderbuffers are *only* renderable.
    - If you won't need to texture from it, prefer to use a renderbuffer.

© Copyright Ian D. Romanick 2007

Texture attachments

- Created as always using `glTexImage2D` et. al.
  - Typically the `pixels` parameter will be `NULL`.
- Different attachment function depending texture dimensionality.
  - `glFramebufferTexture1DEXT` – Attach a 1D texture.
  - `glFramebufferTexture2DEXT` – Attach a 2D texture or a cube map face.
  - `glFramebufferTexture3DEXT` – Attach a slice of a 3D texture.

© Copyright Ian D. Romanick 2007

Renderbuffers

- Created using `glGenRenderbuffersEXT` and `glRenderbufferStorageEXT`.
  - Analogous to `glGenTextures` and `glTexImage2D`.
  - Only way to supply data to a renderbuffer is by rendering to it.
- Attach to FBO using `glFramebufferRenderbufferEXT`.

© Copyright Ian D. Romanick 2007

Dimensions and dimensionality

- The dimensions (i.e., height and width) of all attachments **must** match.
  - This requirement will be relaxed in a future extension.
- The dimensionality (i.e., 1D or 2D) of all attachments must match.
  - A 2D "slice" of a 3D texture is attached, so it is treated as a 2D texture for this purpose.

© Copyright Ian D. Romanick 2007

Questions?

© Copyright Ian D. Romanick 2007

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or the Art Institute of Portland.
- OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.
- Khronos and OpenGL ES are trademarks of the Khronos Group.
- Other company, product, and service names may be trademarks or service marks of others.

© Copyright Ian D. Romanick 2007