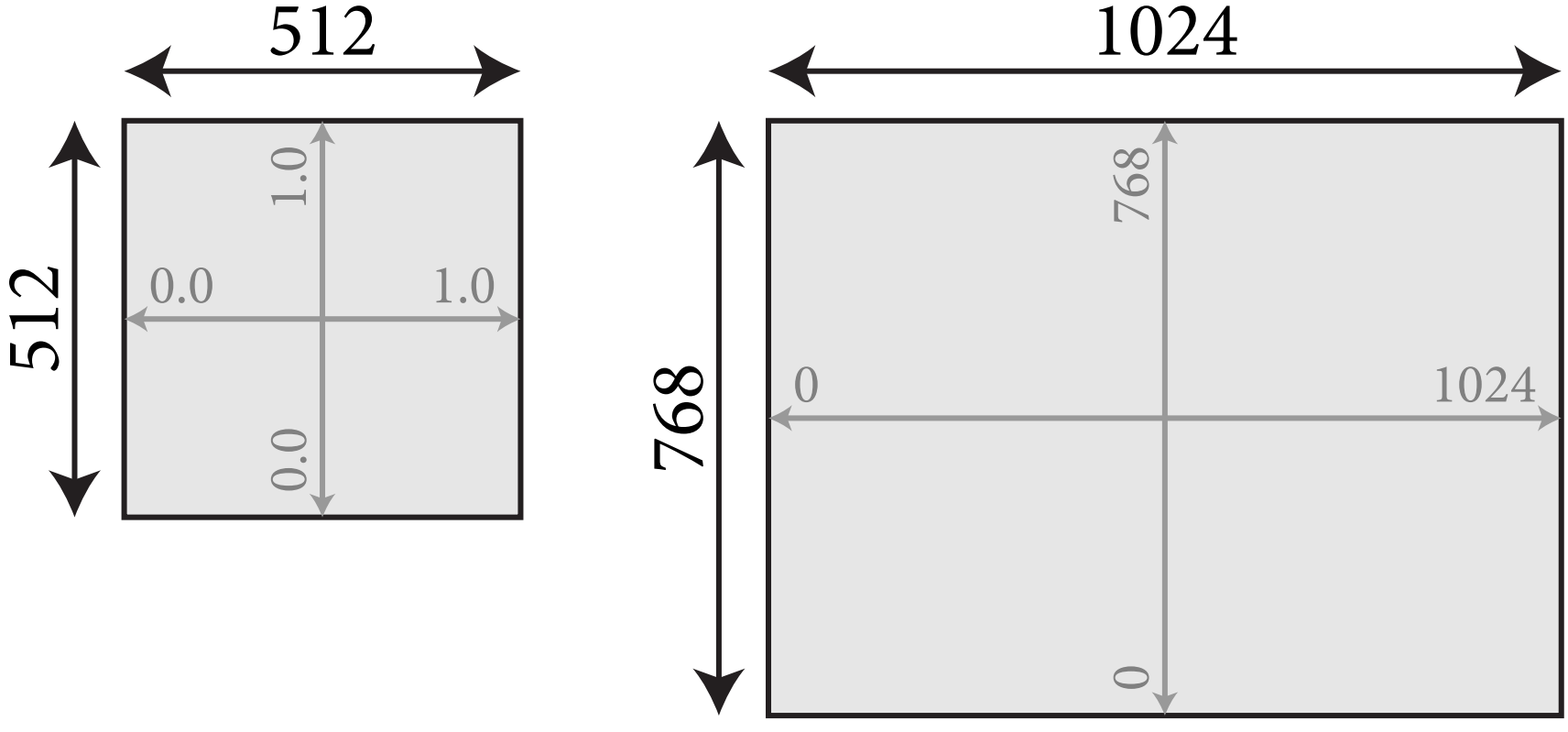
CSC 4356 / ME 4573 Interactive Computer Graphics

Rendering to Textures

“Power-of-two” vs. “rectangular” textures



Rectangular texture API

Usage is exactly like 2D textures, but with a different *target*.

```
GLuint texture;

glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_RECTANGLE, texture);

glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGB,
             w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, p);

glTexParameteri(GL_TEXTURE_RECTANGLE,
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

Rectangular texture GLSL

```
uniform sampler2DRect image;

void main()
{
    gl_FragColor = texture2DRect(image, gl_FragCoord.xy);
}
```

This fragment shader copies a texture to the screen, mapping each texel one-to-one to a pixel.

What is a framebuffer?

A framebuffer is a rendering destination.

1. An array of RGB color pixels
2. An array of depth values, one for each pixel

Until now, we've needed only one framebuffer, but a host of effects become possible when you have multiple…

Framebuffer Objects

A framebuffer object is a GPU-resident OpenGL entity that allows us to manage multiple rendering destinations.

```
GLuint fbo;

glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
```

As usual, we *bind* the object we want to use. In this case, we bind the framebuffer to receive our rendering.

Framebuffer Objects

FBOs are useful because they allow us to attach ordinary texture objects in place of the color and depth buffers.

Let's initialize some rectangular textures. In the following slides assume

```
GLenum target = GL_TEXTURE_RECTANGLE;
```

We already know all about color texture objects…

```
glGenTextures(1, &color);
glBindTexture(target, color);

glTexImage2D(target, 0, GL_RGBA, w, h, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, NULL);

glTexParameteri(target, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(target, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(target, GL_TEXTURE_WRAP_S,     GL_CLAMP_TO_EDGE);
glTexParameteri(target, GL_TEXTURE_WRAP_T,     GL_CLAMP_TO_EDGE);
```

Note the `NULL` pixel data pointer. This texture is uninitialized.

Depth textures are just like any other texture…

```
glGenTextures(1, &depth);
glBindTexture(target, depth);

glTexImage2D(target, 0, GL_DEPTH_COMPONENT24, w, h, 0,
             GL_DEPTH_COMPONENT, GL_UNSIGNED_BYTE, NULL);

glTexParameteri(target, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(target, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(target, GL_TEXTURE_WRAP_S,     GL_CLAMP_TO_EDGE);
glTexParameteri(target, GL_TEXTURE_WRAP_T,     GL_CLAMP_TO_EDGE);
```

…though the formats differ.

Finally, we *attach* our color and depth textures to the framebuffer object.

```
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_COLOR_ATTACHMENT0, target, color, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_DEPTH_ATTACHMENT,  target, depth, 0);
```

Now when we render to this framebuffer, the output appears in our textures. We can subsequently map these textures onto geometry, and re-use what was drawn.

Always be paranoid about errors

```
switch (glCheckFramebufferStatus(GL_FRAMEBUFFER))
{
case GL_FRAMEBUFFER_COMPLETE:                      break;
case GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT:         fbofail("Attachment");
case GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT: fbofail("Missing attachment");
case GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS:         fbofail("Dimensions");
case GL_FRAMEBUFFER_INCOMPLETE_FORMATS:            fbofail("Formats");
case GL_FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER:        fbofail("Draw buffer");
case GL_FRAMEBUFFER_INCOMPLETE_READ_BUFFER:        fbofail("Read buffer");
case GL_FRAMEBUFFER_UNSUPPORTED:                   fbofail("Unsupported");
default:                                           fbofail("Unknown");
}
```

(Note to copy-pasters: you have to implement `fbofail` yourself.)

Good FBO citizenship

Uninitialized textures may contain garbage, so clear them.

```
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

After rendering, revert to the on-screen framebuffer.

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

Don't try to *map* a *bound* texture. As awesome as it sounds, it's not a feedback loop. You'll get useless junk and your output won't port.

Render-to-texture applications

- Reflections
- Refractions
- Shadows
- Physics sim.
- Post-process effects
  - Tone-mapping
  - Motion blur
  - Deferred shading

Much of modern OpenGL rendering technology is built upon render-to-texture.

Example: Water Rendering

Water is both reflective and refractive. We simulate this using FBOs.

1. Render everything *under* the water to a *refraction* texture.
2. Render everything *above* the water to a *reflection* texture, but do so *upside-down*.
3. Render the surface of the water, mixing the reflection and refraction textures following Fresnel's equation.
4. Render the remainder of the scene normally.

Reflection and refraction demo.

Clipping at the surface of the water

We need to slice the **scene** at the plane of the water's surface. Define that plane using the OpenGL built-in clip plane state.

```
double water_plane[4];

water_plane[0] =  0.0;
water_plane[1] = -1.0;
water_plane[2] =  0.0;
water_plane[3] = -0.1;

glClipPlane(GL_CLIP_PLANE0, water_plane);
```

Recall the plane equation: $a\,x + b\,y + c\,z + d\,w = 0$

Clipping at the surface of the water

We do the clip test per-fragment (for portability) so the vertex shader puts the eye-space fragment position in a varying.

```
varying vec4 var_P;

void main()
{
    var_P = gl_ModelViewMatrix * gl_Vertex;
    // ...
    gl_Position = ftransform();
}
```

Clipping at the surface of the water

The fragment shader clipping test is a simple dot product.

Discard the fragment upon failure.

```
varying vec4 var_P;

void main()
{
    if (dot(var_P, gl_ClipPlane[0]) > 0.0)
    {
        // This fragment passes.  Render normally...
    }
    else discard;
}
```

The necessity of clipping demo.

Rendering upside-down

A horizontal reflector flips the scene vertically. To render a reflection, render the scene upside-down. We must flip…

- the light source position,
- all triangle windings,
- all vertex positions,

…across the *plane* of the reflector.

Rendering an upside-down light source

```
GLfloat p[4], q[4];

glGetLightfv(GL_LIGHT0, GL_POSITION, p);
q[0] = +p[0];
q[1] = -p[1];
q[2] = +p[2];
q[3] = +p[3];

glLightfv(GL_LIGHT0, GL_POSITION, q);
{
    // Render the upside-down triangles...
}
glLightfv(GL_LIGHT0, GL_POSITION, p);
```

Rendering upside-down triangles

```
glFrontFace(GL_CW);
{
    // Render the upside-down vertex positions...
}
glFrontFace(GL_CCW);
```

We temporarily change the definition of a triangle's "front" from counter-clockwise to clockwise.

Rendering upside-down vertex positions

```
glPushMatrix();
{
    glTranslated(0.0, +water_plane[3], 0.0);
    glScaled(+1.0, -1.0, +1.0);
    glTranslated(0.0, -water_plane[3], 0.0);

    draw_scene();
}
glPopMatrix();
```

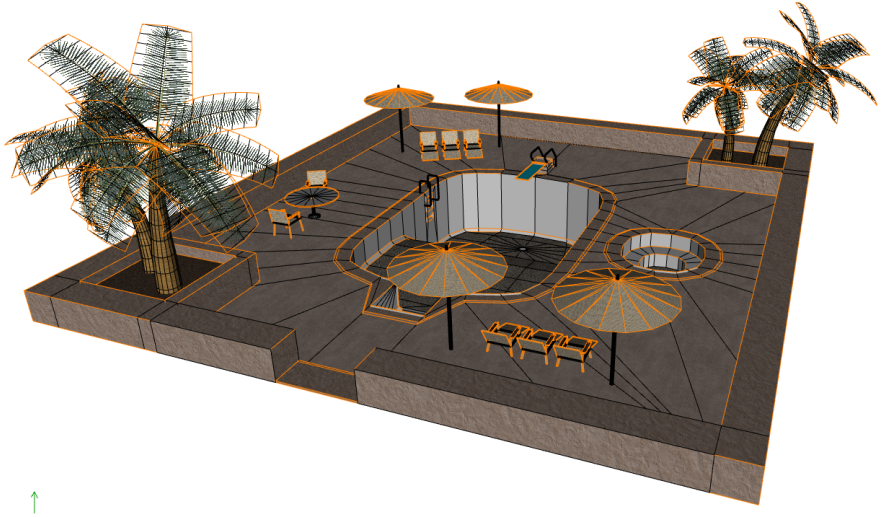
We translate the scene down to the water's surface, flip the Y axis, and translate back to where we started.

Erroneous flipping demo.

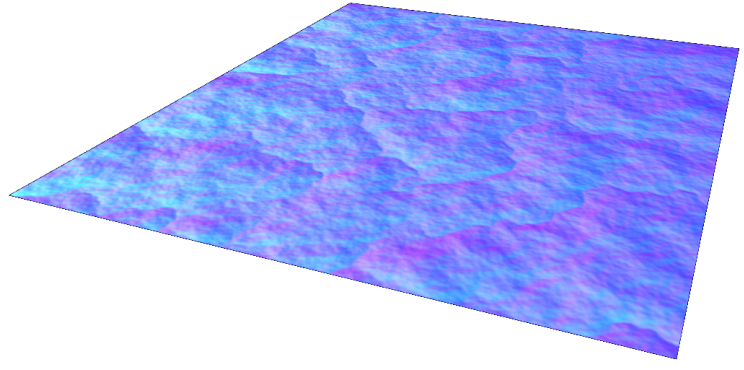
Recap: Water Rendering

1. Render the scene under the water
   to a refraction texture.
2. Render the scene above the water
   to a reflection texture.
3. Render the water using those textures.
4. Render the scene normally.

Recap: Water Rendering



Scene geometry



Water geometry

Step 1 summary: Refraction rendering

1. Bind the refraction texture FBO.
2. Clip away everything above the water.
3. Bind the scene shader.
4. Render the scene geometry.

Step 2 summary: Reflection rendering

1. Bind the reflection texture FBO.
2. Flip the light position.
3. Flip the front face.
4. Flip the vertex transformation.
5. Clip away everything below the water.
6. Bind the scene shader.
7. Render the scene geometry.

Step 3 summary: Water Rendering

1. Bind the null FBO, enabling on-screen rendering.
2. Bind the refraction texture.
3. Bind the reflection texture.
4. Bind the water shader.
5. Render the water geometry.

More about the water shader in a moment…

Step 4 summary: Scene Rendering

1. Bind the scene shader.
2. Render the scene geometry.

Our water is *technically* opaque, so render the scene *last* to ensure that any transparent scene geometry is alpha-blended correctly.

Water Vertex Shader

The vertex shader is trivial, noting only the view vector and texture coordinate varyings.

```
varying vec3 var_V;

void main()
{
    var_V = -(gl_ModelViewMatrix * gl_Vertex).xyz;

    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_Position    = ftransform();
}
```

Trivial Water Fragment Shader

```
uniform sampler2DRect reflection;
uniform sampler2DRect refraction;

varying vec3 var_V;

void main()
{
    vec2 t = gl_FragCoord.xy;

    vec4 R = texture2DRect(refraction, t) * gl_FrontMaterial.diffuse;
    vec4 L = texture2DRect(reflection, t);

    gl_FragColor = vec4((L + R) / 2.0);
}
```

Trivial water demo.

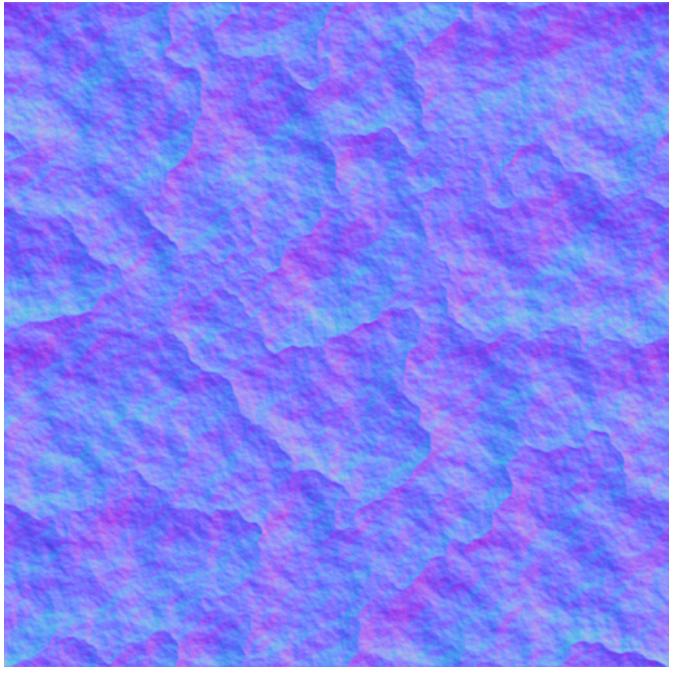
Making Waves

The illusion of rippling water is produced simply by perturbing the reflection and refraction texture coordinate.

This perturbation is driven by a normal map.

This is *not physically correct*, but it looks good.

Making Waves



A rippling animation may be created by combining multiple copies of a single *static* normal map, with each copy independently scaled and translated.

Animated Water Normal Shader

```
uniform sampler2D normal;
uniform float     time;

vec3 water_normal(vec2 p)
{
    vec2 t1 = 0.05 * p + vec2(-0.003,  0.005) * time;
    vec2 t2 = 0.50 * p + vec2(-0.010, -0.020) * time;
    vec2 t3 = 2.20 * p + vec2( 0.014,  0.030) * time;
    vec2 t4 = 5.10 * p + vec2( 0.040, -0.050) * time;

    const vec3 ck = vec3(-2.0, +2.0, -2.0);
    const vec3 cd = vec3(+1.0, -1.0, +1.0);

    return normalize((texture2D(normal, t1).xzy * ck + cd) +
                     (texture2D(normal, t2).xzy * ck + cd) +
                     (texture2D(normal, t3).xzy * ck + cd) +
                     (texture2D(normal, t4).xzy * ck + cd));
}
```

Animated normal demo.

Animated Water Fragment Shader

```
uniform sampler2DRect reflection;
uniform sampler2DRect refraction;
varying vec3 var_V;

void main()
{
    vec3 N = water_normal(gl_TexCoord[0].xy);

    vec2 t = gl_FragCoord.xy + N.xz * 200.0 / length(var_V);

    vec4 R = texture2DRect(refraction, t) * gl_FrontMaterial.diffuse;
    vec4 L = texture2DRect(reflection, t);

    gl_FragColor = mix(R, L, 0.5);
}
```

Animated water demo.

Fresnel Water Fragment Shader

```
varying vec3 var_V;

void main()
{
    vec3 V = normalize(var_V);
    vec3 N = water_normal(gl_TexCoord[0].xy);
    vec3 M = gl_NormalMatrix * N;

    vec2 t = gl_FragCoord.xy + N.xz * 200.0 / length(var_V);

    vec4 R = texture2DRect(refraction, t) * gl_FrontMaterial.diffuse;
    vec4 L = texture2DRect(reflection, t);

    float K = clamp(pow(1.0 - dot(V, M), 2.0), 0.0, 1.0);

    gl_FragColor = mix(R, L, K);
}
```

Fresnel water demo.

Coming up…

Another FBO technique: Shadow mapping.