Preparing for Texture Access

- **These steps are the same when using a shader as when using fixed functionality**
  - Make a specific texture unit active by calling `glActiveTexture`
  - Create a texture object and bind it to the active texture unit by calling `glBindTexture`
  - Set texture parameters by calling `glTexParameter`
  - Define the texture by calling `glTexImage`

- **Not required when using a shader:**
  - Enabling the desired texture on the texture unit by calling `glEnable`
  - Setting the texture function by calling `glTexEnv`

Accessing Texture Maps

- In your shader, declare a uniform variable of type `sampler`
- In your application, call `glUniform1i` to specify the texture unit to be accessed
- From within your shader, call one of the built-in texture functions
  - 1D/2D/3D textures
  - Depth textures
  - Cube maps
  - Projective versions also provided

Vertex Shader Texture Access

- Textures can be accessed from either a fragment shader or a vertex shader
- However, an implementation is allowed to report 0 as the number of supported vertex texture image units
  - Current generation of hardware may report 0
  - Could be a portability issue for some applications
- **Level-of-detail is handled differently:**
  - Some texture calls are allowed only within a vertex shader and express the level-of-detail as an absolute value
  - Other texture calls are allowed only within a fragment shader and the level-of-detail parameter is used to bias the value computed by the graphics hardware
Application Code

```c
static void init2DTexture(GLint texUnit, GLint texName, 
    GLint texWidth, GLint texHeight, 
    GLubyte *texPtr)
{
    glActiveTexture(GL_TEXTURE0 + texUnit);
    glBindTexture(GL_TEXTURE_2D, texName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, texPtr);
    glActiveTexture(GL_TEXTURE0);
}
```

Earth Fragment Shader (1 Texture)

```c
varying float LightIntensity;
uniform sampler2D EarthTexture;

void main (void)
{
    vec3 lightColor = vec3 (texture2D(EarthTexture, glTexCoord[0].st));
    gl_FragColor = vec4 (lightColor * LightIntensity, 1.0);
}
```

Multitexture Example

- Blue Marble images by Reto Stöckli of the NASA/Goddard Space Flight Center
- Put clouds in red component and gloss map in green component of one texture

Multitexture Fragment Shader

```c
uniform sampler2D EarthDay;
uniform sampler2D EarthNight;
uniform sampler2D EarthCloudGloss;

void main (void)
{
    vec2 clouds = texture2D(EarthCloudGloss, TexCoord).rg;
    vec3 daytime = (texture2D(EarthDay, TexCoord).rg * Diffuse + Specular * clouds.g) * (1.0 - clouds.r) + clouds.r * Diffuse;
    vec3 nighttime = texture2D(EarthNight, TexCoord).rg * (1.0 - clouds.r) * 2.0;
    vec3 color = daytime;
    if (Diffuse <= 0.1) color = mix(nighttime, daytime, Diffuse + 0.1) * 5.0;
    gl_FragColor = vec4 (color, 1.0);
}
```
Uses For Texture Memory

- Normals
- Gloss values
- Control values
- Polynomial coefficient values
- Intermediate values from a multipass algorithm
- Lookup tables
- Complex function values
  - Noise
  - Trig functions
- Random numbers
- ???

Procedural Textures

A procedural texture is a texture that is computed in a shader rather than stored in a texture map

Advantages:
- Can be a continuous mathematical function rather than a discrete array of pixel values – therefore infinite precision is possible
- Shader code is likely to be a few kilobytes rather than a few megabytes for a texture map
- Can be paramaterized, allowing a lot of flexibility at run time

Disadvantages:
- Programming skill required (not so for texture maps)
- Texture lookup might be faster than procedural texture computation
- May have aliasing characteristics that are difficult to overcome (texture mapping hardware is built to deal with aliasing issues, e.g., mipmaps)
- Hardware differences may lead to somewhat different appearance on different platforms

Often a hybrid approach will be the right answer
// Stripe Shader - Courtesy Lightwork Design

uniform vec3 LightPosition;
uniform vec3 LightColor;
uniform vec3 EyePosition;
uniform vec3 Specular;
uniform vec3 Ambient;
uniform float Kd;

varying vec3 DiffuseColor;
varying vec3 SpecularColor;

void main(void)
{
  vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
  vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
  vec3 lightVec = normalize(LightPosition - ecPosition);
  vec3 viewVec = normalize(EyePosition - ecPosition);
  vec3 Hvec = normalize(viewVec + lightVec);
  float spec = clamp(dot(Hvec, tnorm), 0.0, 1.0);
  spec = pow(spec, 16.0);
  DiffuseColor = LightColor * vec3 (Kd * dot(lightVec, tnorm));
  DiffuseColor = clamp(Ambient + DiffuseColor, 0.0, 1.0);
  SpecularColor = clamp((LightColor * Specular * spec), 0.0, 1.0);
  gl_TexCoord[0] = gl_MultiTexCoord0;
  gl_Position     = ftransform();
}

uniform vec3 StripeColor;
uniform vec3 BackColor;
uniform float Width;
uniform float Fuzz;
uniform float Scale;
uniform float Threshold;
uniform vec3 SurfaceColor;

void main (void)
{
  float scaled_t = fract(gl_TexCoord[0].t * Scale);
  float frac1 = clamp(scaled_t / Fuzz, 0.0, 1.0);
  float frac2 = clamp((scaled_t - Width) / Fuzz, 0.0, 1.0);
  frac1 = frac1 * (1.0 - frac2);
  frac1 = frac1 * frac1 * (3.0 - (2.0 * frac1));
  vec3 finalColor = mix(BackColor, StripeColor, frac1);
  finalColor = finalColor * DiffuseColor + SpecularColor;
  gl_FragColor = vec4 (finalColor, 1.0);
}

uniform vec3 DiffuseColor;
uniform vec3 SpecularColor;

void main (void)
{
  float ss = fract(gl_TexCoord[0].s * Scale.s);
  float tt = fract(gl_TexCoord[0].t * Scale.t);
  if ((ss > Threshold.s) && (tt > Threshold.t)) discard;
  vec3 finalColor = SurfaceColor * DiffuseColor + SpecularColor;
  gl_FragColor = vec4 (finalColor, 1.0);
}
# Dimple Vertex Shader

```glsl
varying vec3 LightDir;
varying vec3 EyeDir;
uniform vec3 LightPosition;
attribute vec3 Tangent;

void main() {
    EyeDir = vec3(gl_ModelViewMatrix * gl_Vertex);
    gl_Position = ftransform();
    gl_TextureCoord[0] = gl_MultiTexCoord0;

    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 t = normalize(gl_NormalMatrix * Tangent);
    vec3 b = cross(n, t);

    vec3 v;
    v.x = dot(LightPosition, t);
    v.y = dot(LightPosition, b);
    v.z = dot(LightPosition, n);
    LightDir = normalize(v);

    v.x = dot(EyeDir, t);
    v.y = dot(EyeDir, b);
    v.z = dot(EyeDir, n);
    EyeDir = normalize(v);
}
```

# Dimple Fragment Shader

```glsl
varying vec3 LightDir;
varying vec3 EyeDir;

uniform vec3 SurfaceColor; // = (0.7, 0.6, 0.18)
uniform float BumpDensity; // = 16.0
uniform float BumpSize; // = 0.15
uniform float SpecularFactor; // = 0.5

void main (void) {
    vec3 litColor;
    vec2 c = BumpDensity * gl_TextureCoord[0].st;
    vec2 p = fract(c) - vec2(0.5);
    float d, f;
    d = p.x * p.x + p.y * p.y;
    f = 1.0 / sqrt(d + 1.0);
    if (d >= BumpSize)
        { p = vec2(0.0); f = 1.0; }
    vec3 normDelta = vec3(p.x, p.y, 1.0) * f;
    litColor = SurfaceColor * max(dot(normDelta, LightDir), 0.0);
    vec3 reflectDir = reflect(LightDir, normDelta);
    float spec = max(dot(EyeDir, reflectDir), 0.0);
    spec *= SpecularFactor;
    litColor = min(litColor + spec, vec3(1.0));
    gl_FragColor = vec4(litColor, 1.0);
}
```
Procedural Shader Demo