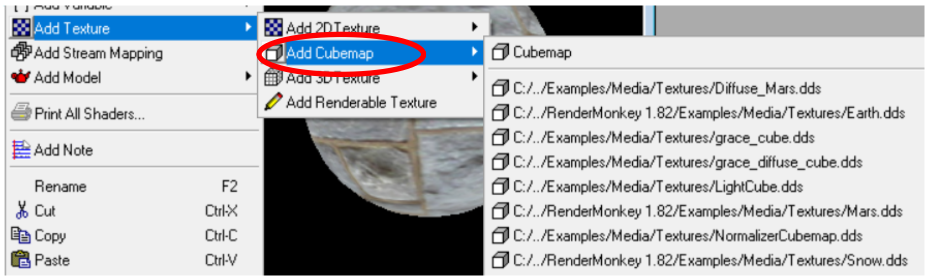
This lab exercises consist of three parts: Cube mapping, particle systems and texture-based animations. You may find some effects to be implemented in this lab are essential for some effects required in doing your ACW.

Part 1. Cube Mapping

Exercise 1: Draw the environment

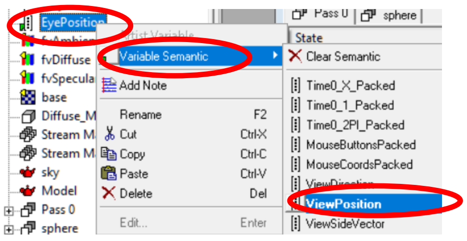
1. Start from the default OpenGL Position effect.
2. Add a cubemap texture to the effect.


Add Texture
Add Stream Mapping
Add Model
Print All Shaders...
Add Note
Rename F2
Cut Ctrl-X
Copy Ctrl-C
Paste Ctrl-V
Add 2D Texture
Add Cubemap
Add 3D Texture
Add Renderable Texture
Cubemap
C:/../Examples/Media/Textures/Diffuse_Mars.dds
C:/../RenderMonkey 1.82/Examples/Media/Textures/Earth.dds
C:/../Examples/Media/Textures/grace_cube.dds
C:/../Examples/Media/Textures/grace_diffuse_cube.dds
C:/../Examples/Media/Textures/LightCube.dds
C:/../RenderMonkey 1.82/Examples/Media/Textures/Mars.dds
C:/../Examples/Media/Textures/NormalizerCubemap.dds
C:/../RenderMonkey 1.82/Examples/Media/Textures/Snow.dds

3. Rename pass 0 as Environment; add the loaded cube map as a ***texture object*** to the pass and rename it as "Sky"
4. Open vertex shader program and add following variables:

```
uniform vec4 EyePosition;
varying vec3 vDir;
```

5. Set the ***semantics*** of EyePosition to be ViewPosition.



6. Define the view direction vDir in the vertex shader:

```
… …
vDir = gl_Vertex.xyz;
vec3 Pos2beViewed = EyePosition.xyz + vDir ;
gl_Position = gl_ModelViewProjectionMatrix * vec4(Pos2beViewed, 1.0);
… …
```

7. In the fragment shader, specify the fragment colour using the colour sampled from the cubemap corresponding to the view direction vDir:

```
uniform samplerCube Sky;
varying vec3 vDir;
void main(void)
{
        gl_FragColor = textureCube( Sky, vDir);
}
```

8. Configure the render state and specify the culling mode to be NONE or FRONT. Save and recompile your shader programs.

Exercise 2: Draw reflective environment mapped object

1. Add a default pass as the second pass to your effect and rename it as MyShinyObject.
2. Add the cubemap ***texture object*** you used in the pass for drawing the environment.
3. Similar to the way you implement per-fragment lighting, add necessary variables to both the vertex shader and fragment shader to find the reflection view direction in the fragment shader. For example,

```
uniform vec4 EyePosition;
varying vec3 Normal;
varying vec3 vDir;
void main(void)
{
        gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
        vDir = EyePosition.xyz-gl_Vertex.xyz;
        Normal = gl_Normal;
        … …
}
```

4. The fragment shader of the pass should be similar to the one in the first pass (see point 6 in Exercise 1), except that you should use the reflected view direction to fetch the colour from the cubemap texture.
    a) Pass through view direction and normal vector from the vertex shader to the fragment shader:
    ```
    uniform samplerCube Sky;
    varying vec3 vDir;
    varying vec3 Normal;
    ```
    b) In main( ), find the reflected view direction and use it to get a colour from the cube map:

    ```
    vec3 reflectView=reflect(-vDir, Normal);
    gl_FragColor = textureCube( Sky, reflectView);
    ```

5. Edit the render states of the two passes to be able to view the shiny object properly: **disable** the depth test for the pass drawing the environment and **enable** the depth test in pass for drawing the shiny object. The cull mode may also need to be changed.
6. Save your programs and recompile.

Part 2. Particle Systems Based Fire Animation

1. You can start doing the exercise from a default sample effect provided in RenderMonkey, such as the default "Textured" GLSL effect. Instead of using a collection of particles, you can begin with just one particle by loading a single quad mode, say, ScreenAlignedQuad.3ds model.
2. Open vertex shader:
    a. Scale the quad into proper size:
    ```
    vec4 bbPos = gl_Vertex;
    bbPos.xy *= billboardSize;
    ```
    b. Do billboarding. First get the inverse of the view matrix by declare a 4x4 matrix and set its semantics to be ViewInverse.
    ```
    uniform mat4  viewInv;
    ```
    c. Then reset of the orientation of each quad using the first two columns of the matrix:
    ```
    … …
    //Billboarding:
    vec3 Vx= viewInv [0].xyz;
    vec3 Vy= viewInv [1].xyz;

    bbPos.xyz =  (bbPos.x * Vx+ bbPos.y * Vy).xyz;
    ```
    d. Translate the quad to a required position, say, by a vector (20.0, 20.0, 20.0):
    ```
    bbPos.xyz +=vec3(20.0);
    ```
    e. Transform the quad into clipping space:
    ```
    gl_Position = gl_ModelViewProjectionMatrix* bbPos;
    ```
    f. Save and recompile your shader programs. Change your view direction, you should see the quad always facing the view direction.

3. Now let's consider how to make the quad in motion by replacing vec3(20.0) specified in step d as a function of time.

   1) Add a timer to the vertex shader in a similar way as you implement the pulsating effect:
      ```
      uniform float time;
      ```
   2) Specify an initial shooting velocity **v** for the object.
      ```
      uniform vec3 v;
      ```
   3) Specify the frequency and lifecycle of a particle.
      ```
      uniform float freq;
      uniform float lifecycle;
      ```
   4) Translate the billboard's position with velocity **v**:
      ```
      float T=mod(freq *time, lifecycle);
      bbPos.xyz +=v*T;
      ```
   5) Transform the quad into clipping space:
      ```
      gl_Position = gl_ModelViewProjectionMatrix* bbPos;
      ```

4. Fade the colour of the object. It should be brightest at the time of its birth and completely dark at the time of its death. For instance, you can define the fading rate in the vertex shader in the following way and pass it to the fragment shader to modify the particle colour:
   ```
   fadingRate = (1.0 -  T/lifecycle);
   ```
5. Compile and run your GLSL programs. If the quad is properly billboarded and animated, you can now change the single quad model with QuadArray model provided in Rendermonkey, which provides 100 rectangles differed by its **z** values.
6. Texture mapping the quads. You may need to define the texture coordinates in the vertex shader to map the texture properly, this is because the reqired texture coordinates have not been provided in the QuadArray model:
   ```
   Texcoord = sign(inPos.xy)*0.5 + 0.5;
   ```
   You may also need to configure the cull mode to be 'NONE' so that both sides of each quad are visible.
7. Reposition the quads within the array and define a function to specify how you want to shoot them (Refer to my lecture notes to get an idea on how to distribute these quads using their z-coordinate).

Part 3. Texture Processing

Exercise 1. Implement texture transformation

Start from a default OpenGL **Screen-Aligned Quad** effect. Implement various texture transformation operations, such as translation, scaling, shearing, and rotation. Similar to performing per-vertex transformations, per-fragment transformations can also be implemented either uniformly or locally.

Exercise 2. Texture based animation

1. Start from a default OpenGL **Screen-Aligned Quad** effect.
2. Add a timer to the vertex shader in a similar way as you implement the pulsating effect:
```
uniform float time;
```
3. Specify a way to alter the incoming texture coordinate, for example, to make the texture texels move with velocity **v** by altering the texture coordinates in the following way
```
uniform vec2 v;

… …

void main(void)
{
    gl_FragColor = texture2D( Texture0, texCoord + v* time);
}
```
4. (Optional) Use the idea in step 3 to animate a real world phenomenon, such as rain or snow.

Exercise 3. Texture based fire animation

Refer to the lecture notes for detailed description of the technique.

Exercise 4. Image smoothing

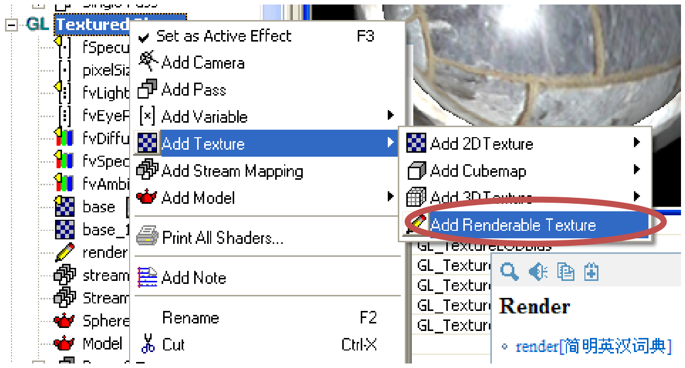
1. Starting from the default GLSL Screen-Aligned Quad effect.
2. Change the texture with the one you want to process.
3. 2D image processing is done in per-fragment manner. The smooth algorithm used to process a given image is implemented in fragment shader. In fragment shader, define pixel size of the image:
```
uniform float pixelSize;
```
4. For each fragment, find the average colour of its neighbouring fragment colours, which can be of different forms. The average colour of square

neighbouring fragments of size 5x5 can be found, for example, from the following function:

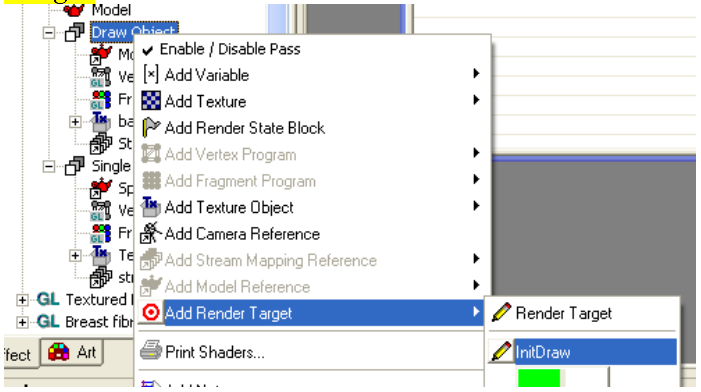
```
vec4 texSmoothing(sampler2D Texture0, vec2 uv)
{
  vec4 smoothedCol=vec4(0.0);
  for (int i=-2; i<=2; i++){
    for (int j=-2; j<=2; j++){
      smoothedCol += texture2D(Texture0,  uv + vec2(i, j)*pixelSize );
    }
  }

  return smoothedCol/25.0;
}
```

Exercise 5 Glow effects

1. Starting from a default GLSL effect or your own per-fragment lighting effect.
2. Render the effect into a texture:
   a) Add a renderable texture to the effect by clicking Add Renderable Texture. Rename the name of the renderable texture as required, say, initDraw.



   b) Select the pass which you would like to render the image into a texture rather than to the screen. To render the effect into a texture, add the previously added renderable texture in the pass by clicking Add Render Target:



3. Smooth the rendered image.
   a) Add a new pass to smooth the image you have generated in the first pass using the method implemented in Exercise 4:

b) Blend the image generated in the first pass with the smoothed image.

Exercise 6. A burning object

In Exercise 5, further apply the texture-based fire animation technique (Exercise 3) to the smoothed image to make an object burning. (Hint: In texture-based fire animation, replace the image representing the fire shape with the smoothed image).

==

Written by Dr Qingde Li(q.li@hull.ac.uk).
Mar. 25, 2019.