Tutorial 14 : Render To Texture

Render-To-Texture is a handful method to create a variety of effects. The basic idea is that you render a scene just like you usually do, but this time in a texture that you can reuse later.

Applications include in-game cameras, post-processing, and as many GFX as you can imagine.

Render To Texture

We have three tasks : creating the texture in which we're going to render ; actually rendering something in it ; and using the generated texture.

Creating the Render Target

What we're going to render to is called a Framebuffer. It's a container for textures and an optional depth buffer. It's created just like any other object in OpenGL :

```
// The framebuffer, which regroups 0, 1, or more textures, and 0 or 1
depth buffer.
GLuint FramebufferName = 0;
glGenFramebuffers(1, &FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

Now we need to create the texture which will contain the RGB output of our shader. This code is very classic :

```
// The texture we're going to render to
GLuint renderedTexture;
glGenTextures(1, &renderedTexture);

// "Bind" the newly created texture : all future texture functions
will modify this texture
glBindTexture(GL_TEXTURE_2D, renderedTexture);
```

```
// Give an empty image to OpenGL ( the last "0" )
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, 1024, 768, 0,GL_RGB,
GL_UNSIGNED_BYTE, 0);

// Poor filtering. Needed !
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

We also need a depth buffer. This is optional, depending on what you actually need to draw in your texture; but since we're going to render Suzanne, we need depth-testing.

```
// The depth buffer
GLuint depthrenderbuffer;
glGenRenderbuffers(1, &depthrenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthrenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 1024, 768);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
GL_RENDERBUFFER, depthrenderbuffer);
```

Finally, we configure our framebuffer

```
// Set "renderedTexture" as our colour attachement #0
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
renderedTexture, 0);

// Set the list of draw buffers.
GLenum DrawBuffers[1] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, DrawBuffers); // "1" is the size of DrawBuffers
```

Something may have gone wrong during the process, depending on the capabilities of the GPU. This is how you check it :

```
// Always check that our framebuffer is ok
if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE)
return false;
```

Rendering to the texture

Rendering to the texture is straightforward. Simply bind your framebuffer, and draw your scene as usual. Easy !

```
// Render to our framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

```
glViewport(0,0,1024,768); // Render on the whole framebuffer, complete
from the lower left corner to the upper right
```

The fragment shader just needs a minor adaptation :

```
layout(location = 0) out vec3 color;
```

This means that when writing in the variable “color”, we will actually write in the Render Target 0, which happens to be our texure because DrawBuffers[0] is GL_COLOR_ATTACHMENT*i*, which is, in our case, *renderedTexture*.

To recap :

- *color* will be written to the first buffer because of layout(location=0).
- The first buffer is GL_COLOR_ATTACHMENT0 because of DrawBuffers[1] = {GL_COLOR_ATTACHMENT0}
- GL_COLOR_ATTACHMENT0 has *renderedTexture* attached, so this is where your color is written.
- In other words, you can replace GL_COLOR_ATTACHMENT0 by GL_COLOR_ATTACHMENT2 and it will still work.

Note : there is no layout(location=i) in OpenGL < 3.3, but you use glFragData[i] = mvvalue anyway.

Using the rendered texture

We’re going to draw a simple quad that fills the screen. We need the usual buffers, shaders, IDs, …

```
// The fullscreen quad's FBO
GLuint quad_VertexArrayID;
glGenVertexArrays(1, &quad_VertexArrayID);
glBindVertexArray(quad_VertexArrayID);

static const GLfloat g_quad_vertex_buffer_data[] = {
    -1.0f, -1.0f, 0.0f,
    1.0f, -1.0f, 0.0f,
    -1.0f,  1.0f, 0.0f,
    -1.0f,  1.0f, 0.0f,
    1.0f, -1.0f, 0.0f,
    1.0f,  1.0f, 0.0f,
};

GLuint quad_vertexbuffer;
```

```
glGenBuffers(1, &quad_vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, quad_vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_quad_vertex_buffer_data),
g_quad_vertex_buffer_data, GL_STATIC_DRAW);

// Create and compile our GLSL program from the shaders
GLuint quad_programID = LoadShaders( "Passthrough.vertexshader",
"SimpleTexture.fragmentshader" );
GLuint texID = glGetUniformLocation(quad_programID, "renderedTexture");
GLuint timeID = glGetUniformLocation(quad_programID, "time");
```

Now you want to render to the screen. This is done by using 0 as the second parameter of glBindFramebuffer.

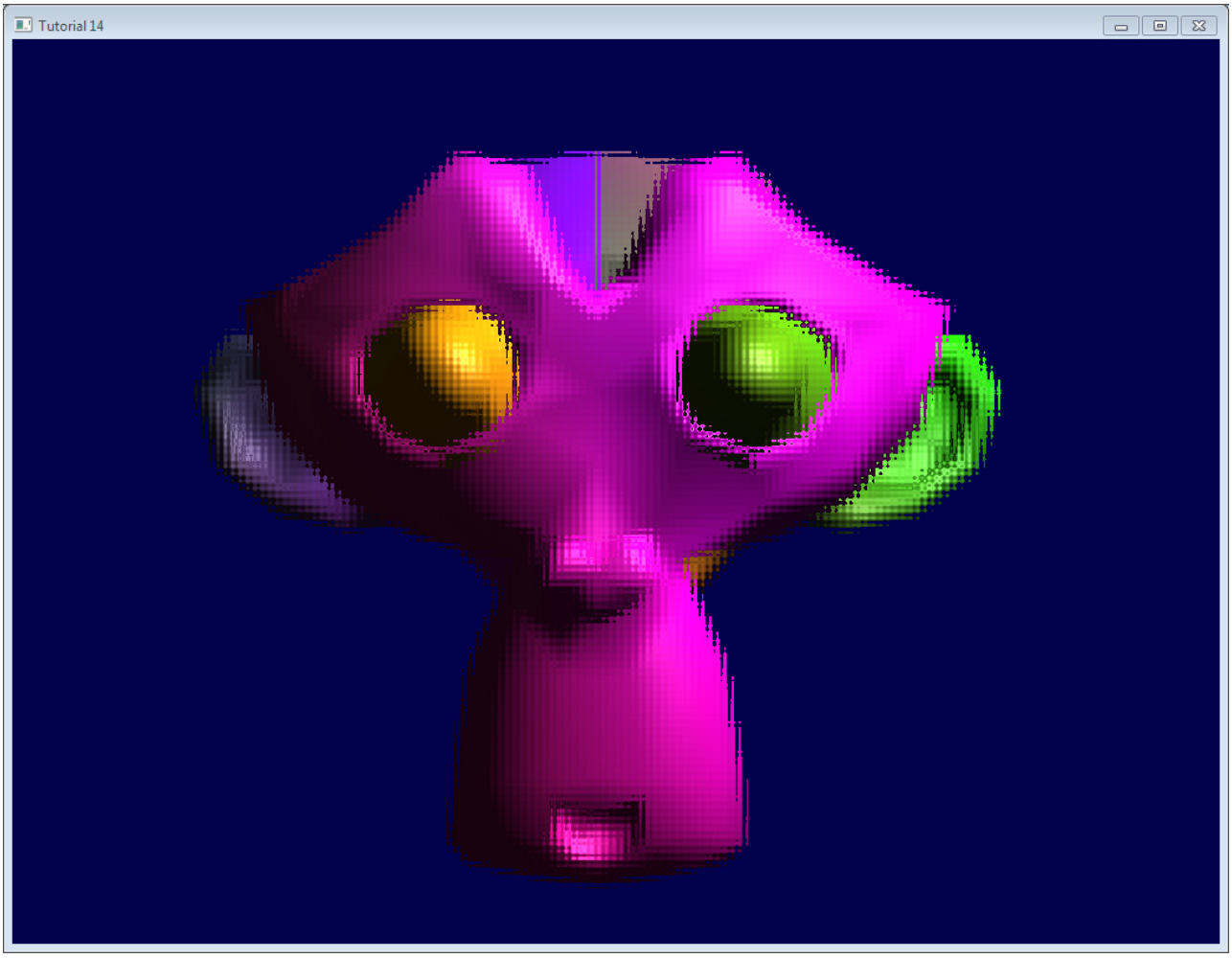
```
// Render to the screen
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0,0,1024,768); // Render on the whole framebuffer, complete
from the lower left corner to the upper right
```

We can draw our full-screen quad with such a shader:

```
#version 330 core

in vec2 UV;

out vec3 color;

uniform sampler2D renderedTexture;
uniform float time;

void main(){
    color = texture( renderedTexture, UV + 0.005*vec2( sin(time+1024.0*U
}
```

This code simply sample the texture, but adds a tiny offset which depends on time.

Results



Going further

Using the depth

In some cases you might need the depth when using the rendered texture. In this case, simply render to a texture created as follows :

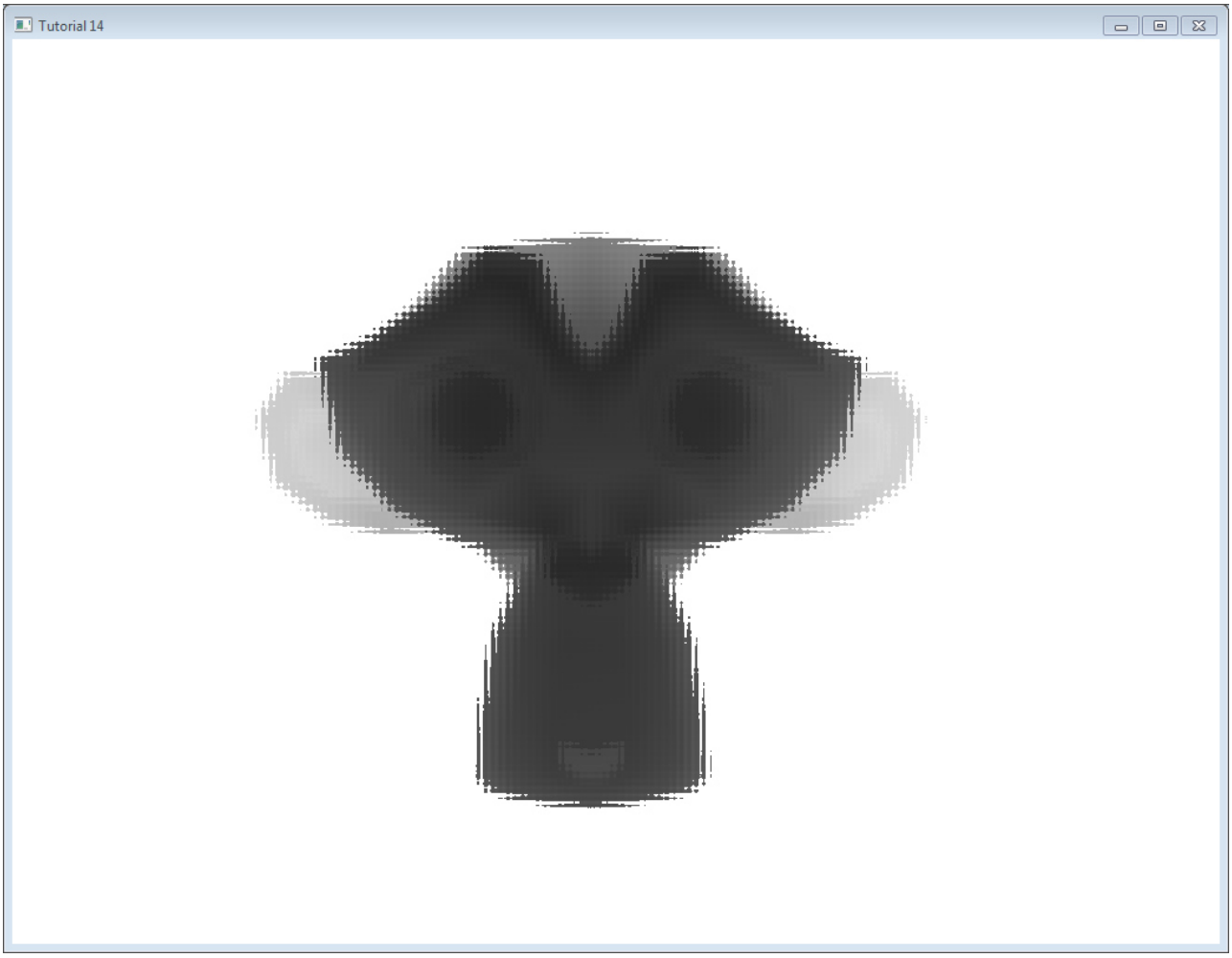
```
glTexImage2D(GL_TEXTURE_2D, 0,GL_DEPTH_COMPONENT24, 1024, 768,
0,GL_DEPTH_COMPONENT, GL_FLOAT, 0);
```

("24" is the precision, in bits. You can choose between 16, 24 and 32, depending on your needs. Usually 24 is fine)

This should be enough to get you started, but the provided source code implements this too.

Note that this should be somewhat slower, because the driver won't be able to use some optimisations such as Hi-Z.

In this screenshot, the depth levels are artificially "prettified". Usually, its much more difficult to see anything on a depth texture. Near = Z near 0 = black, far = Z near 1 = white.



Multisampling

You can write to multisampled textures instead of “basic” textures : you just have to replace glTexImage2D by glTexImage2DMultisample in the C++ code, and sampler2D/texture by sampler2DMS/texelFetch in the fragment shader.

There is a big caveat, though : texelFetch needs another argument, which is the number of the sample to fetch. In other words, there is no automatic “filtering” (the correct term, when talking about multisampling, is “resolution”).

So you may have to resolve the MS texture yourself, in another, non-MS texture, thanks to yet another shader.

Nothing difficult, but it’s just bulky.

Multiple Render Targets

You may write to several textures at the same time.

Simply create several textures (all with the correct and same size !), call glFramebufferTexture

with a different color attachement for each, call glDrawBuffers with updated parameters (something like (2,{GL_COLOR_ATTACHMENT0,GL_COLOR_ATTACHMENT1}})), and add another output variable in your fragment shader :

```
layout(location = 1) out vec3 normal_tangentspace; // or whatever
```

Hint : If you effectively need to output a vector in a texture, floating-point textures exist, with 16 or 32 bit precision instead of 8… See glTexImage2D's reference (search for GL_FLOAT).

Hint2 : For previous versions of OpenGL, use glFragData[1] = myvalue instead.

Exercices

- Try using glViewport(0,0,512,768); instead of glViewport(0,0,1024,768); (try with both the framebuffer and the screen)
- Experiment with other UV coordinates in the last fragment shader
- Transform the quad with a real transformation matrix. First hardcode it, and then try to use the functions of controls.hpp ; what do you notice ?

contact@opengl-tutorial.org