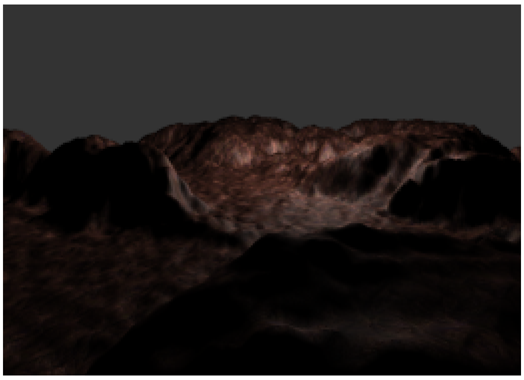
Tutorial 12: Real-Time Lighting B



Summary

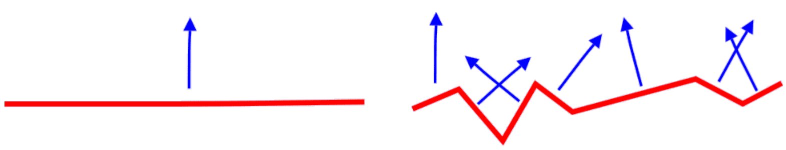
The last tutorial taught you the basics of real time lighting, including using the *normal* of a surface to calculate the *diffusion* and *specularity.* Surfaces are seldom completely flat, however, so to increase the realism of our graphical rendering, we must find a way to simulate the roughness of a surface. This tutorial will introduce *bump mapping*, a way of storing the roughness of a surface, and how to use the bump map in our lighting shaders.

New Concepts

Bump maps, Tangent Space, Tangent Generation

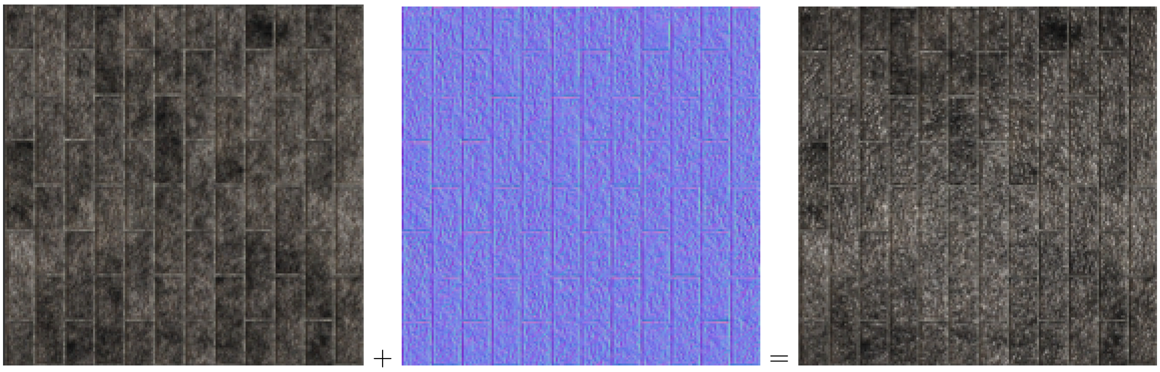
Bump Maps

In the previous tutorial, it was mentioned that even in *Phong* shading, surface normal information must be determined via interpolation of vertices. Although this will create a unique normal for every fragment, the normals still won't accurately represent the material you are trying to light. Take a wall made out of brick, for example. We can model such a wall using a single quad, resulting in a single normal (or a slightly spherical distribution of normals in cases where indexing is used). In reality, a brick wall is not flat at all, but made up out of many subsurfaces - the mortar that adheres the bricks is likely to be slightly receded, and the bricks are likely to have quite a rough surface. So, how to accurately simulate the roughness of a surface? We *could* use progressively more and more polygons to subdivide the surface, but such a solution would be prohibitively computationally expensive. A better solution is to use a *bump map.* Much as the texture maps we've been using so far simulate the *colour* of a surface, a bump map simulates its *roughness.* Instead of a *colour* per texel, a bump map stores a *normal* per texel, each of which points in a varying direction according the relative bumpiness of the material they are simulating.



Left: A surface and its normal Right: A Surface with normals derived from a bump map

Bump maps are generally saved as a simple RGB texture, with each channel storing the x, y and z axis of a normalised direction vector, respectively. As bumpmaps are colour data like any other texture, interpolation can be performed on the sampled data, resulting in unique per-fragment normals no matter how large the bump mapped surface is on screen.

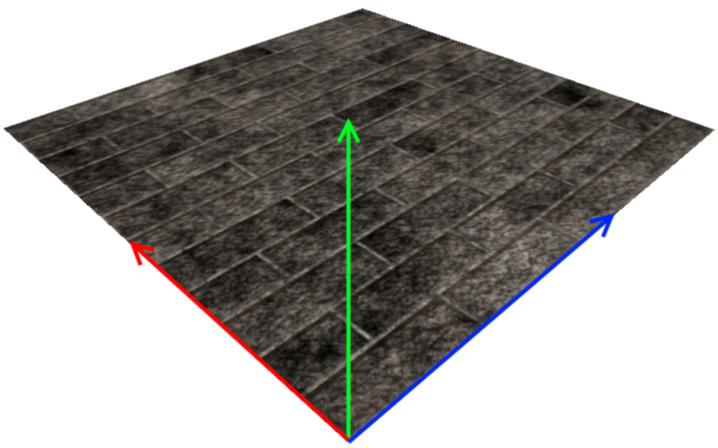


An example of a texture, its bump map, and the result of per fragment lighting using them

Tangents

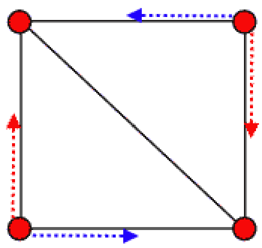
Bump maps define their normals in what is called *tangent space.* In order to use them in lighting calculations, they must be transformed into some other space, such as world space. The bump map may be used for lots of different geometry, applied in an entirely arbitrary manner - the texture coordinates could have been flipped, rotated or stretched, and the geometry itself could be at any orientation, so unless we have a unique bump map for *every* triangle, and those triangles *never move*, we'll have to do some extra processing on the bump map's normals to make them usable.

We can't just transform a bump map by the surface or vertex normal, either - the normal is only a single axis, and there are infinitely many orientations about a single axis. In order to fully transform a tangent space normal into world space, we need *3* axis, forming a rotation matrix. The surface normal is one of these axis', so what are the other two? The direction axis we know about is the normal, which is orthogonal to the surface, so it makes sense that the extra direction we need runs along, or is *tangent to*, the surface of the polygon we want to bump map. The remaining axis, the *binormal*, can be calculated by taking the cross product of the normal and tangent direction vectors - remember, the result of a cross product is a vector *orthogonal* to its input.



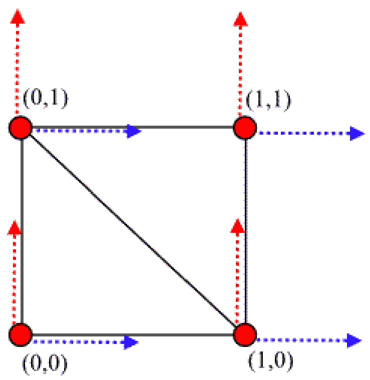
Green = Normal, Blue = Tangent, Red = Binormal

So, for each vertex, we need a direction vector that points along the surface of the triangle it makes up, but *which direction* along the surface? If you followed how to generate normals in the last tutorial, you might think that one of the direction vectors (c-a or b-a) that point along a triangle's edge would do it. Unfortunately, this won't work. See the example below, where a quad made of two triangles ends up with conflicting tangent vectors - tangent space normals in the top right triangle would be transformed in the opposite direction to those in the bottom left!



Blue = tangent direction. Red = binormal direction

Instead, to derive the tangent vector, we use the *positive x-axis* direction of the surface's *texture* coordinates. Doing so keeps the tangent vectors consistent - as long as the texture coordinates are consistent!



Blue = tangent direction. Red = binormal direction

Tangent space to World space

To perform the tangent-space to world-space conversion, the normalised tangent, binormal, and normal vectors are combined into a 'Tangent Binormal Normal' rotation matrix, like so:

$$\begin{bmatrix} Tx & Bx & Nx \\ Ty & By & Ny \\ Tz & Bz & Nz \end{bmatrix}$$

With the binormal being formed from the normal and tangent:

$$Binormal = Cross(Normal,\ Tangent)$$

The final step is to multiply the bump map normal by the TBN matrix, transforming it from tangent space into world space, where it can be used to perform lighting calculations:

$$World\ Normal = TBN \cdot Tangent\ normal$$

After this, the tangent space normals will have been transformed into world space, suitable for plugging into the lighting equations you learnt in the last tutorial.

Example Program

The example program for this tutorial is going to build on the previous tutorial's - only this time our terrain will have bump mapping applied, giving us far more realistic lighting! We don't need any new classes, but we do need to modify our *Mesh* class to support bump mapping, as well as the *Renderer* class we made. In your *Shaders* folder, we need to add two new files, bumpVertex.glsl and bumpFragment.glsl - we'll be using the vertex shader again in the next tutorial.

Mesh Class

Header File

Like the normals we introduced in the previous tutorial,tangents are specified at the vertex level, so we need a new *attribute*. Like other attributes, this is as simple as adding a new named constant to the *MeshBuffer* **enum**:

```
enum MeshBuffer {
    VERTEX_BUFFER, COLOUR_BUFFER, TEXTURE_BUFFER,
    NORMAL_BUFFER, TANGENT_BUFFER,INDEX_BUFFER,  //Tangents!
    MAX_BUFFER
};
```

Mesh.h

In the *Mesh* class itself, we need another new **protected** member variable - a **pointer** to some *Vector3* data, storing our mesh's tangents. We also need a **protected** function, *GenerateTangents*, to generate them - with an additional helper function called *GenerateTangent*. Finally, we need to add the **protected** member variable *bumpTexture* to store the OpenGL name of the mesh bump map, and some **public** accessors for it.

```
class Mesh  {
...
public:
void      SetBumpMap(GLuint tex)  {bumpTexture = tex;}
GLuint    GetBumpMap()            {return bumpTexture;}

protected:
...
void      GenerateTangents();
Vector3   GenerateTangent(const Vector3 &a, const Vector3 &b,
                          const Vector3 &c, const Vector2 &ta,
                          const Vector2 &tb,const Vector2 &tc);

Vector3*        tangents;
GLuint          bumpTexture;
...
}
```

Mesh.h

Class File

In our **constructor**, we should make our new *tangents* pointer have a **NULL** value, and in the **destructor**, we should **delete** it. We also do the same for our *bumpTexture* OpenGL name.

```
Mesh::Mesh(void)  {
...
tangents        = NULL;
bumpTexture     = 0;
...
```

Mesh.cpp

```
Mesh::~Mesh(void) {
...
delete[]tangents;
glDeleteTextures(1,&bumpTexture); //Just like the texture map...
...
```

renderer.cpp

As with normals in the previous tutorial, we need to modify the *BufferData* function slightly, so that tangents, if they exist, are stored in graphics memory:

```
void  Mesh::BufferData()   {
...
   if(tangents) {
      glGenBuffers(1, &bufferObject[TANGENT_BUFFER]);
      glBindBuffer(GL_ARRAY_BUFFER, bufferObject[TANGENT_BUFFER]);
      glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector3),
                      tangents, GL_STATIC_DRAW);
      glVertexAttribPointer(TANGENT_BUFFER,3, GL_FLOAT, GL_FALSE,0,0);
      glEnableVertexAttribArray(TANGENT_BUFFER);
   }
...
```

Mesh.cpp

We need to make a little change to the *Draw* function of the *Mesh* class. We need to **bind** the bump map texture before we draw the mesh geometry! It's done in just the same way as the diffuse texture, but is bound to texture unit *1*, rather than *0*. The new *Draw* function should look like this:

```
void Mesh::Draw() {
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_2D, texture);

   glActiveTexture(GL_TEXTURE1);                   //New!!!
   glBindTexture(GL_TEXTURE_2D, bumpTexture);    //New!!!

   glBindVertexArray(arrayObject);
   if(bufferObject[INDEX_BUFFER]) {//Added by the index buffers tut...
      glDrawElements(type, numIndices, GL_UNSIGNED_INT, 0);
   }
   else{
      glDrawArrays(type, 0, numVertices);
   }
   glBindVertexArray(0);
}
```

Mesh.cpp

Finally, we need a function to generate the tangent data for each vertex. *GenerateTangents* works in a pretty similar way to the *GenerateNormals* function we wrote last tutorial. We want to loop around every vertex, and generate the tangent for it. As with normals, the tangent of a vertex that is part of more than one face is the *normalised sum* of all of the face tangents. As with normal generation, we account for meshes that modify their data, by allocating the memory for the tangets if necessary, and resetting all tangents to zero.

```
void Mesh::GenerateTangents() {
   if(!tangents) {
      tangents = new Vector3[numVertices];
   }
   if(!texCoords) {
      return; //Can't use tex coords if there aren't any!
   }
   for(GLuint i = 0; i < numVertices; ++i){
      tangents[i] = Vector3();
   }

   if(indices) {
      for(GLuint i = 0; i < numIndices; i+=3){
         int a = indices[i];
         int b = indices[i+1];
         int c = indices[i+2];

         Vector3 tangent = GenerateTangent(vertices[a],vertices[b],
                           vertices[c],textureCoords[a],
                           textureCoords[b],textureCoords[c]);

         tangents[a] += tangent;
         tangents[b] += tangent;
         tangents[c] += tangent;
      }
   }
   else{
      for(GLuint i = 0; i < numVertices; i+=3){
         Vector3 tangent = GenerateTangent(vertices[i],vertices[i+1],
                           vertices[i+2],textureCoords[i],
                           textureCoords[i+1],textureCoords[i+2]);

         tangents[i]   += tangent;
         tangents[i+1] += tangent;
         tangents[i+2] += tangent;
      }
   }
   for(GLuint i = 0; i < numVertices; ++i){
      tangents[i].Normalise();
   }
}
```

Mesh.cpp

Whether the *Mesh* has indices or not, we use the helper function *GenerateTangent* to calculate the actual tangent. It takes in 6 parameters - the three positions and texture coordinates that make up the triangle we're generating the tangents for. Like normals, we calculate the vectors (c-a) and (b-a), also doing the same with the texture coordinates of the triangle. From that, we can work out which local space orientation corresponds to the x-axis in texture space (the *axis* variable) and determine which way is the positive direction (the *factor* variable, which will flip *axis* if necessary).

```
Vector3 Mesh::GenerateTangent(const Vector3 &a,const Vector3 &b,
                              const Vector3 &c,const Vector2 &ta,
                              const Vector2 &tb,const Vector2 &tc) {
   Vector2 coord1  = tb-ta;
   Vector2 coord2  = tc-ta;

   Vector3 vertex1 = b-a;
   Vector3 vertex2 = c-a;

   Vector3 axis = Vector3(vertex1*coord2.y - vertex2*coord1.y);

   float factor = 1.0f / (coord1.x * coord2.y - coord2.x * coord1.y);

   return axis * factor;
}
```

Mesh.cpp

Shader Class

As we're adding a new vertex attribute, we need a way of binding that attribute to a shader input. As with all of the other new attributes, we do so in the *Shader* class *SetDefaultAttributes* function:

```
void  Shader::SetDefaultAttributes()   {
   glBindAttribLocation(program, VERTEX_BUFFER,  "position");
   glBindAttribLocation(program, COLOUR_BUFFER,  "colour");
   glBindAttribLocation(program, NORMAL_BUFFER,  "normal");
   glBindAttribLocation(program, TANGENT_BUFFER, "tangent");//New ;)
   glBindAttribLocation(program, TEXTURE_BUFFER, "texCoord");
}
```

Shader.cpp

HeightMap Class

Like last tutorial, we need to modify our HeightMap class, to call our new *GenerateTangents* function, right after the *GenerateNormals* function we added last time.

```
HeightMap::HeightMap(std::string name) {
...
   GenerateNormals();
   GenerateTangents();

   BufferData();
...
```

HeightMap.cpp

Renderer Class file

We need to modify the Renderer class we made last tutorial a little bit, to load in the bump map for our landscape, and our new shaders. We begin by modifying the **constructor**, which needs to load in the new shader files.

```
#include "Renderer.h"
Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
   camera       = new Camera(0.0f,0.0f,Vector3(
   RAW_WIDTH*HEIGHTMAP_X / 2.0f,500,RAW_HEIGHT*HEIGHTMAP_Z));

   heightMap      = new HeightMap(TEXTUREDIR"terrain.raw");
   currentShader  = new Shader(SHADERDIR"BumpVertex.glsl",
                        SHADERDIR"BumpFragment.glsl");
```

renderer.cpp

Then, we set the heightmap to have a diffuse and bump map texture - the latter using the new *SetBumpMap* function we just added to the *Mesh* class. Note how the bump map is loaded just like the diffuse textures you've been using so far in this tutorial series - it's all just data, we'll just use it in a different way in fragment shader. We then have an **if** statement to make sure the shader can link, and that the heightmap has the textures required. If this is the case, we set both the diffuse and bump textures of the heightmap to repeat.

```
   heightMap->SetTexture(SOIL_load_OGL_texture(
                TEXTUREDIR"Barren Reds.JPG", SOIL_LOAD_AUTO,
                SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));

   heightMap->SetBumpMap(SOIL_load_OGL_texture(
                TEXTUREDIR"Barren RedsDOT3.JPG", SOIL_LOAD_AUTO,
                SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));

   if(!currentShader->LinkProgram() ||
      !heightMap->GetTexture() || !heightMap->GetBumpMap() ) {
      return;
   }
   SetTextureRepeating(heightMap->GetTexture(),true);
   SetTextureRepeating(heightMap->GetBumpMap(),true);
```

renderer.cpp

We end our constructor in the same way as last tutorial, by initialising the *Light*, the projection matrix, and by enabling depth testing.

```
   light = new Light(Vector3((RAW_HEIGHT*HEIGHTMAP_X / 2.0f),
                500.0f,(RAW_HEIGHT*HEIGHTMAP_Z / 2.0f)),
                Vector4(1,1,1,1), (RAW_WIDTH*HEIGHTMAP_X) / 2.0f);

   projMatrix = Matrix4::Perspective(1.0f,15000.0f,
                 (float)width / (float)height, 45.0f);

   glEnable(GL_DEPTH_TEST);
   init = true;
}
```

renderer.cpp

We need to make one little change to the *RenderScene* function of last tutorial's *Renderer* class - on line 108, we set the current shader's **uniform** variable *bumpTex* to be texture unit 1. This matches up to the changes we made to the *Mesh* class *Draw* function, which binds its bump map to texture unit 1.

```
38 void Renderer::RenderScene()  {
39    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
40
41    glUseProgram(currentShader->GetProgram());
42    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
43                                    "diffuseTex"), 0);
44    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
45                                    "bumpTex"), 1);
46
47    glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
48                      "cameraPos"),1,(float*)&camera->GetPosition());
49
50    UpdateShaderMatrices();
51    SetShaderLight(*light);
52
53    heightMap->Draw();
54
55    glUseProgram(0);
56    SwapBuffers();
57 }
```

renderer.cpp

Vertex Shader

The vertex shader we're going to write is very similar to the one in the previous tutorial. This time though, we have another new input attribute, *tangent*, and two new output attributes, *tangent* and *binormal*. Like normals, tangents are transformed by the normal matrix, which we do on line 29. Then, on line 30, we do the same for the result of the cross product of the normal and the tangent, creating the binormal vector. Other than that, it's the same as before, so you should know what everything else does by now.

```
1  #version 150 core
2  uniform mat4 modelMatrix;
3  uniform mat4 viewMatrix;
4  uniform mat4 projMatrix;
5  uniform mat4 textureMatrix;
6
7  in  vec3 position;
8  in  vec4 colour;
9  in  vec3 normal;
10 in  vec3 tangent; //New!
11 in  vec2 texCoord;
12
13 out Vertex {
14    vec4 colour;
15    vec2 texCoord;
16    vec3 normal;
17    vec3 tangent;  //New!
18    vec3 binormal; //New!
19    vec3 worldPos;
20 } OUT;
21
```

```
void main(void)        {
   mat3 normalMatrix = transpose(inverse(mat3(modelMatrix)));

   OUT.colour        = colour;
   OUT.texCoord      = (textureMatrix * vec4(texCoord, 0.0, 1.0)).xy;

   OUT.normal        = normalize(normalMatrix * normalize(normal));
   OUT.tangent       = normalize(normalMatrix * normalize(tangent));
   OUT.binormal      = normalize(normalMatrix *
                       normalize(cross(normal,tangent)));

   OUT.worldPos      = (modelMatrix * vec4(position,1)).xyz;

   gl_Position       = (projMatrix * viewMatrix * modelMatrix) *
                        vec4(position, 1.0);
}
```

bumpvertex.glsl

Fragment Shader

As with the vertex shader, our new fragment shader is adapted from the previous tutorial. Only now instead of using an interpolated vertex normal in the lighting calculations, we're going to use a transformed tangent space normal from a bump map. We start off by adding a new texture sampler, *bumpTex*, and adding the *tangent* and *binormal* vertex attributes to the vertex input block.

```
1  #version 150 core
2
3  uniform sampler2D diffuseTex;
4  uniform sampler2D bumpTex; //New!
5
6  uniform vec3   cameraPos;
7  uniform vec4   lightColour;
8  uniform vec3   lightPos;
9  uniform float  lightRadius;
10
11 in Vertex {
12    vec3 colour;
13    vec2 texCoord;
14    vec3 normal;
15    vec3 tangent;  //New!
16    vec3 binormal; //New!
17    vec3 worldPos;
18 } IN;
19
20 out vec4 fragColour;
```

bumpfragment.glsl

Then, in our **main** function, we begin by creating the TBN matrix we need to transform the tangent space normal to world space. On line 24, you can see how GLSL allows the use of 3 **vec3**s as input variables into a **mat3**. Then, on line 26, we sample the bump map, and transform the resulting *vec3* by the TBN matrix, giving us a world space *normal* variable. Now, you might be wondering why we multiply the vec3 by 2.0 and then subtract 1.0. Well, as you should know by now, the axis of a vec3 runs from -1.0 to 1.0 - but the input sampled from a texture runs from 0.0 to 1.0! So, to convert the sample into the correct space, we must multiply it by 2.0 (giving us a space from 0.0 to 2.0), and then subtract 1.0 (giving us a space from -1.0 to 1.0).

```
void main(void)   {
    vec4 diffuse        = texture(diffuseTex, IN.texCoord);
    //New!
    mat3 TBN            = mat3(IN.tangent, IN.binormal, IN.normal);
    //New!
    vec3 normal         = normalize(TBN * (texture(bumpTex,
                                    IN.texCoord).rgb * 2.0 - 1.0));
```

bumpfragment.glsl

Once we've done that, we can just use the sampled normal instead of the vertex normal, in all of the same calculations as in the last tutorial.

```
    vec3  incident      = normalize(lightPos - IN.worldPos);
    float lambert       = max(0.0, dot(incident, normal));  //Different!

    float dist          = length(lightPos - IN.worldPos);
    float atten         = 1.0 - clamp(dist / lightRadius, 0.0, 1.0);

    vec3 viewDir        = normalize(cameraPos - IN.worldPos);
    vec3 halfDir        = normalize(incident + viewDir);

    float rFactor       = max(0.0, dot(halfDir, normal));   //Different!
    float sFactor       = pow(rFactor, 33.0 );

    vec3 colour         =  (diffuse.rgb * lightColour.rgb);
    colour              += (lightColour.rgb * sFactor) * 0.33;
    fragColour      =   vec4(colour * atten * lambert, diffuse.a);
    fragColour.rgb      += (diffuse.rgb * lightColour.rgb) * 0.1;
}
```

bumpfragment.glsl

Tutorial Summary

Now when you run last tutorial's example program, you should see the same fully lit landscape, but the lighting, and in particular the specular highlights, should look far more realistic. With the basics of lighting out of the way, in the next tutorial we'll look at adding *environment maps* to your scenes, creating realistic skies and reflections as we do so.

Further Work

1) Surfaces can have separate diffuse and specular lighting components - this can make a surface look iridescent, like a Beetle's carapace. How would you add a separate specular colour component? What changes would be made to the shaders, and to vertex attributes?

2) In the last tutorial, there was brief mention of gloss maps - a way of defining the specular power on a per pixel level. Try making a gloss map for the ground texture used for the height map - remember, the higher the value, the 'tighter' the re ections will appear! How many channels of a texture would be used? What could go in the others?