MVD: Advanced Graphics 1

18 - Material Sets, Normal and Specular maps

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull

Switching between forward and deferred

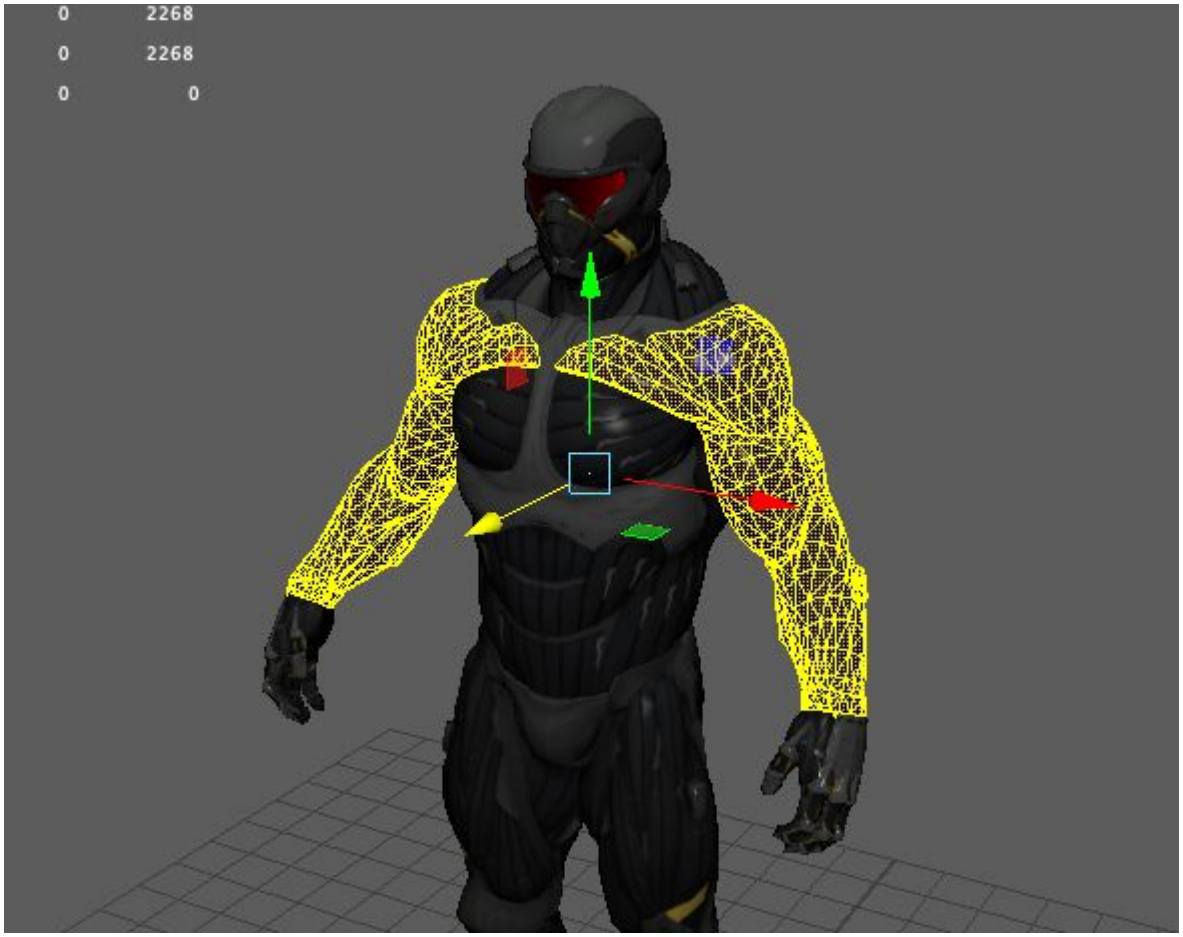
Mesh component switch:

```
enum RenderMode {
    RenderModeForward = 0,
    RenderModeDeferred = 1
};

struct Mesh : public Component {
    int geometry;
    int material;
    RenderMode render_mode;
};
```

Slight difference in lighting because our deferred renderer has no ambient light!

Material sets



Material sets

Use same vertices

but draw a subset of faces with different material

Why not just use meshes and submeshes? Not optimal, storing vertices multiple times

But this isn't optimal either as we are switching materials within each object - should *definitely* try to not switch shader

e.g in OBJ file + MTL file

```
usemtl Leg
s 1
f 53/29/53 54/30/53 55/31/54
f 56/32/55 55/31/54 57/30/56
f 58/33/57 59/34/58 60/29/59
f 60/29/59 59/34/58 57/30/56
f 61/35/60 57/30/56 59/34/58
f 57/30/56 61/35/60 56/32/55
f 62/36/61 56/32/55 61/35/60
f 56/32/55 62/36/61 63/37/62
f 64/38/63 56/32/55 63/37/62
f 65/39/64 64/38/63 63/37/62
```

```
newmtl Leg
Ns 96.078431
Ka 0.000000 0.000000 0.000000
Kd 0.940000 0.940000 0.940000
Ks 0.500000 0.500000 0.500000
Ni 1.000000
d 1.000000
illum 2
map_Kd leg_dif.tga
map_Bump leg_showroom_ddn.tga
map_Ks leg_showroom_spec.tga
```

Changes to geometry (in GraphicsUtilities.h)

```
std::vector<int> material_sets; //each entry is upper limit of set, in TRIANGLES
std::vector<int> material_set_ids; //each entry is id of material for material set

void render(int set); //new render function - render only certain amount of faces
```

Creating materials sets - given for you

Function in Geometry (already given)

Receives upper triangle count, and material id

```
void Geometry::createMaterialSet(int tri_count, int material_id) {
    material_sets.push_back(tri_count);
    material_set_ids.push_back(material_id);
}
```

Geometry render set - need to write this

Don’t forget material_sets contains top triangle to draw

```
void Geometry::render(int set) {
    //bind the vao
    glBindVertexArray(vao);
    //if first set, draw from start to "end of set 0" (* 3 to convert from triangles
    //to indices)
    if (set == 0)
        glDrawElements(GL_TRIANGLES, material_sets[set] * 3, GL_UNSIGNED_INT, 0);
    else {
        //start triangle is end triangle of previous set
        GLuint start_index = material_sets[set - 1] * 3;
        //end triangle is end of current set
        GLuint end_index = material_sets[set] * 3;
        //count is the number of indices to draw
        GLuint count = end_index - start_index;

        glDrawElements(GL_TRIANGLES, //things to draw
                       count, //number of indices
                       GL_UNSIGNED_INT, //format of indices
                       (void*)(start_index * sizeof(GLuint))); //pointer to start!
    }
    glBindVertexArray(0);
}
```

Create Materials & Geometry

New functions:

Parsers::parseMTL

- reads mtl file and adds materials to graphics system, with name

GraphicsSystem:::createMultiGeometryFromFile

- reads multi face-set .obj file and creates material sets for geometry
- overrides material set in cpp, uses material name set in .obj file

Rendering - renderMeshComponent

First deal with case where there are no materials sets - just render basic geometry

```
//draw raw geom if no material sets
if (geom.material_sets.size() == 0)
    geom.render();
else {
    //draw material sets
}
```

Rendering

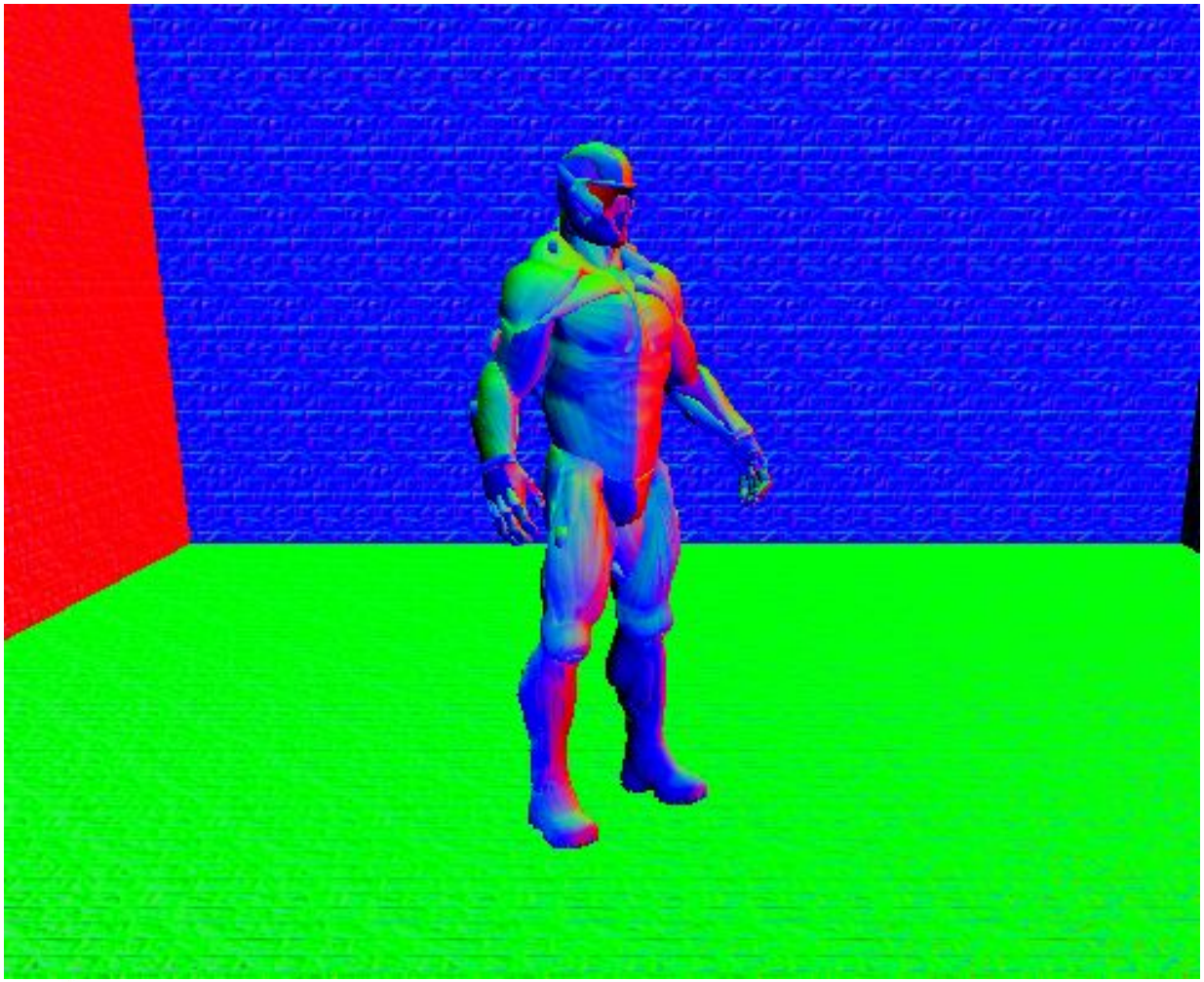
If material sets, change uniform and render set

```
//draw raw geom if no material sets
if (geom.material_sets.size() == 0)
    geom.render();
else {
    //loop material sets
    for (int i = 0; i < geom.material_sets.size(); i++) {
        //set current material id of set
        current_material_ = geom.material_set_ids[i];
        setMaterialUniforms();
        //render current set
        geom.render(i);
    }
}
```

TASK

Load multimaterial object, get it working :)

Normal mapping



Bump/Normal mapping

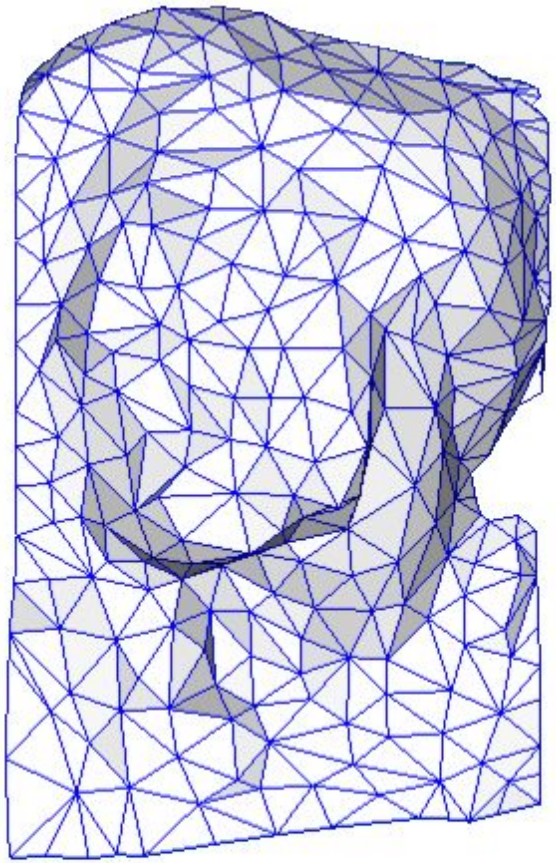
Move surface normals at fragment level

i.e normal doesn't only depend on model geometry

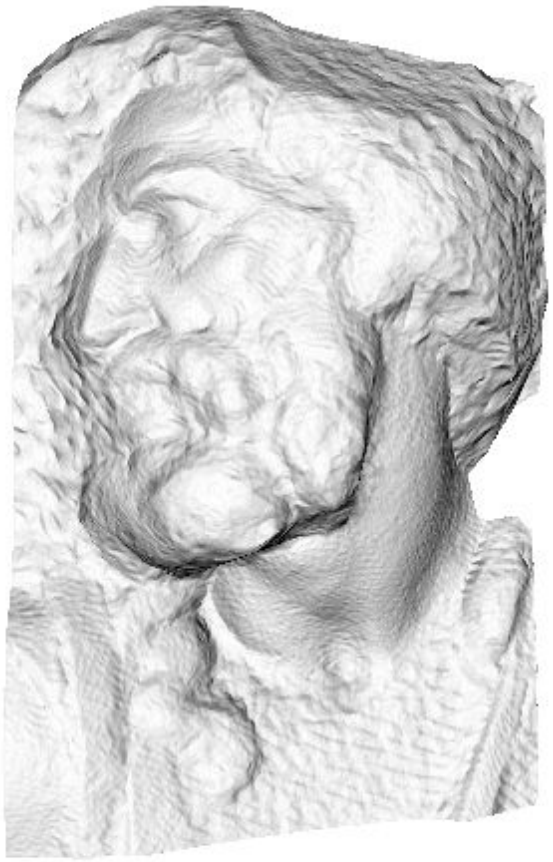
Bump/Normal Mapping

Permits massive reduction in geometry, keeping visual detail

original mesh
4M triangles



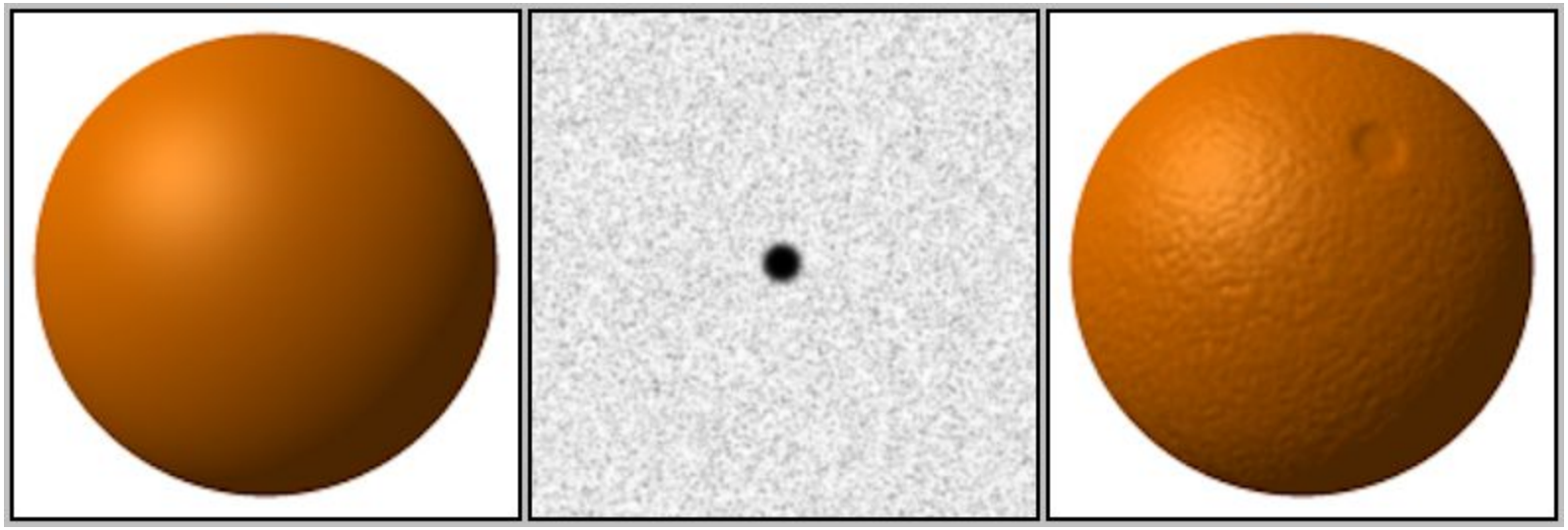
simplified mesh
500 triangles



simplified mesh
and normal mapping
500 triangles

Bump mapping

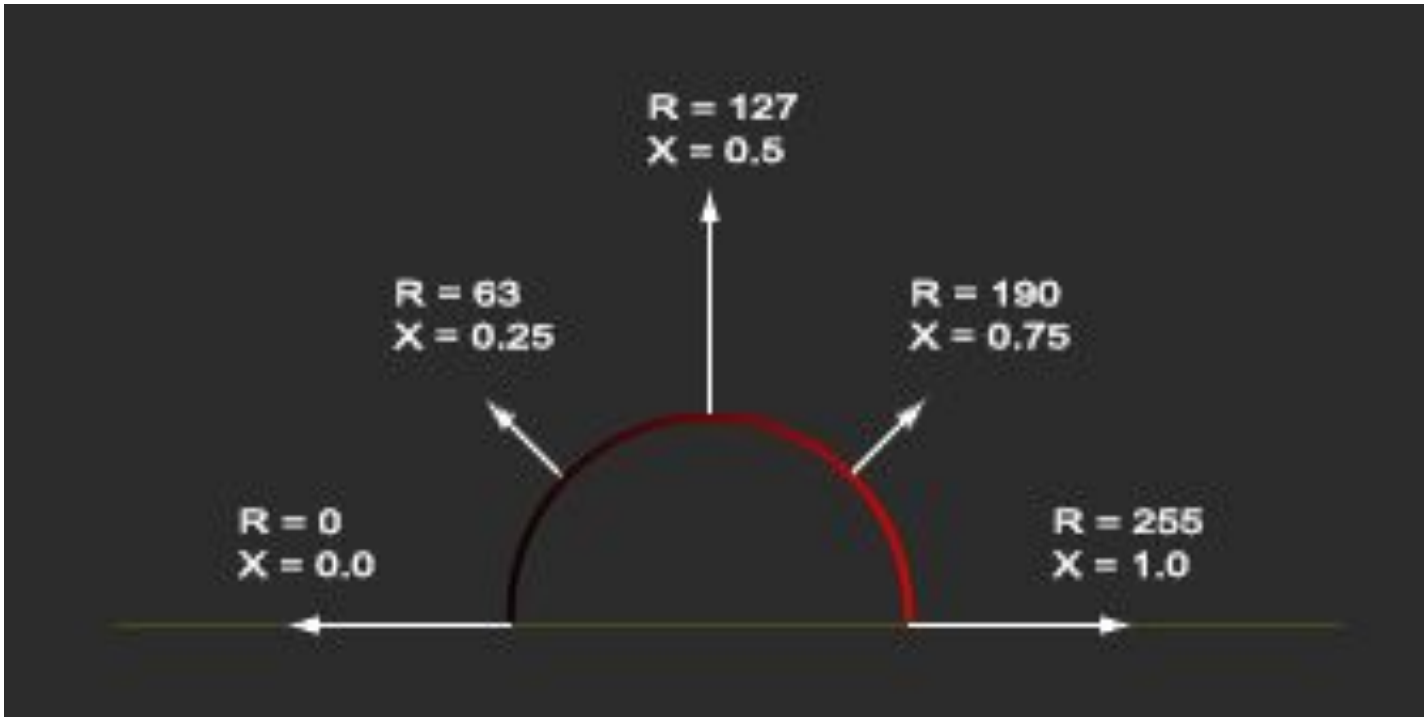
Takes gradient (1st derivative) of greyscale image



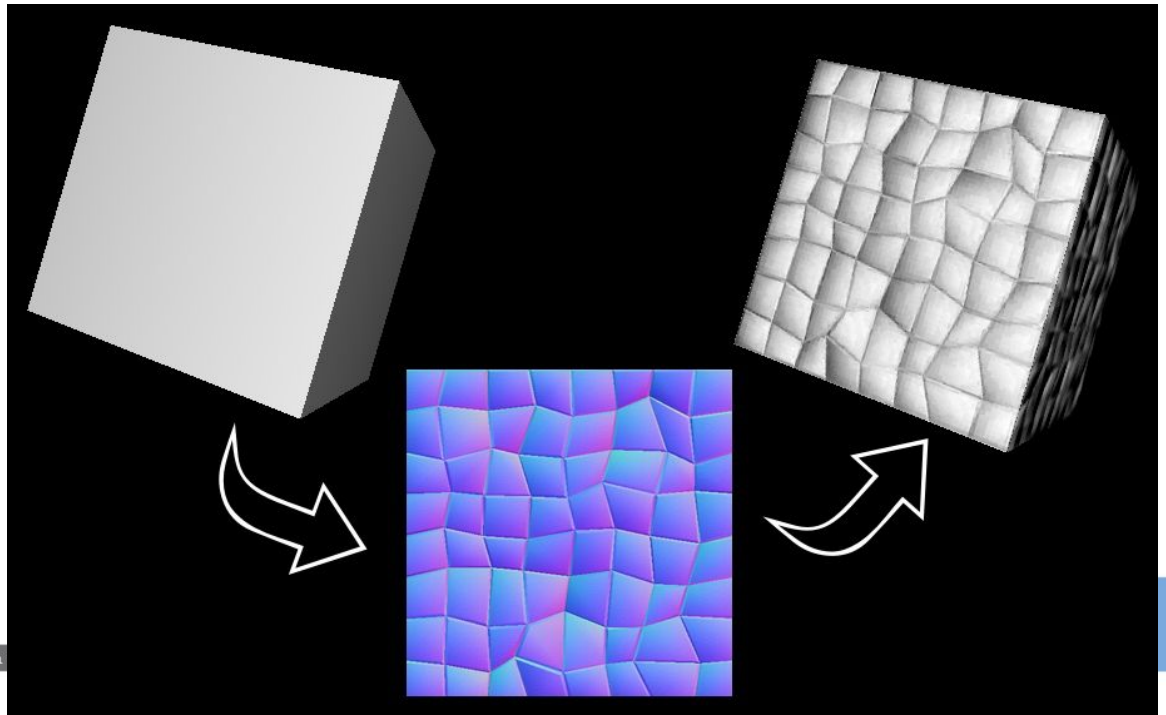
Normal mapping

Encodes the normal direction into the pixel of image

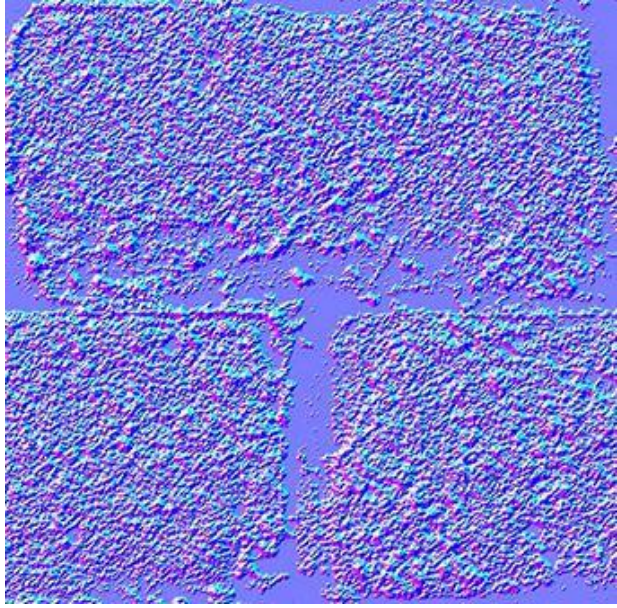
e.g. using red channel for x; green for y; and blue for z



Normal Mapping



Normal mapping

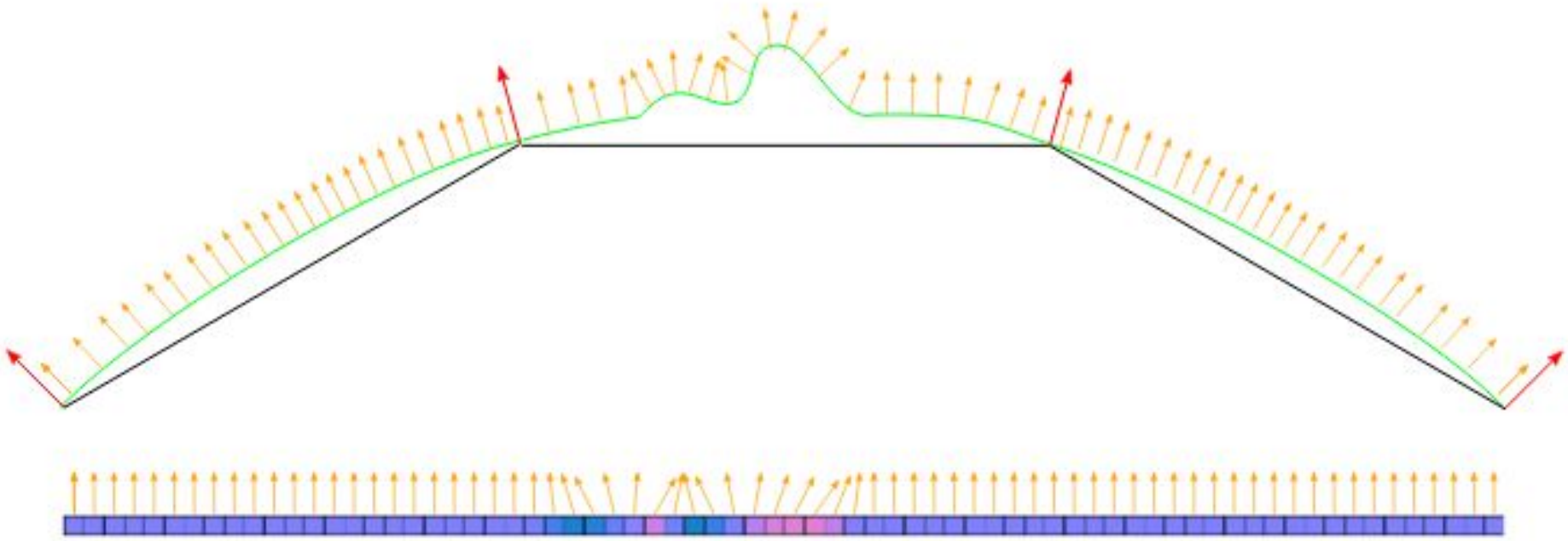


Encode x,y,z values of normal into a rgb values texture

In each RGB texel is encoded a XYZ vector : each colour component is between 0 and 1, and each vector component is between -1 and 1, so this simple mapping goes from the texel to the normal:

```
normal = (2*color)-1 // on each component
```

Normal maps modify the pixel normal

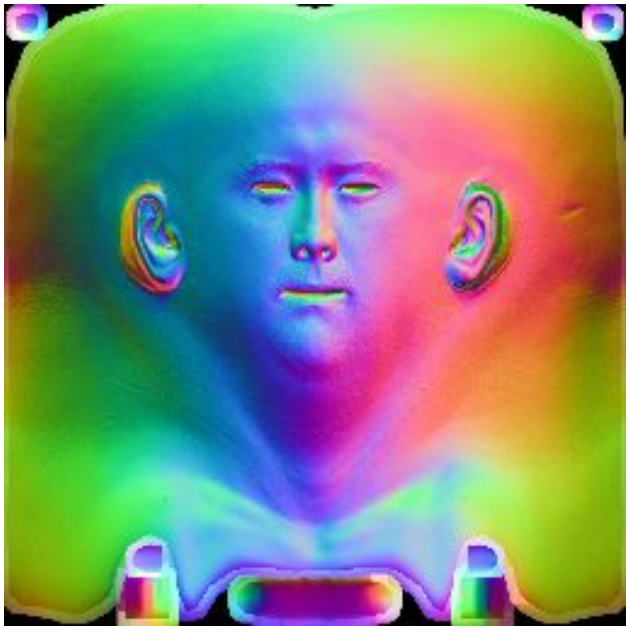


normal map texture

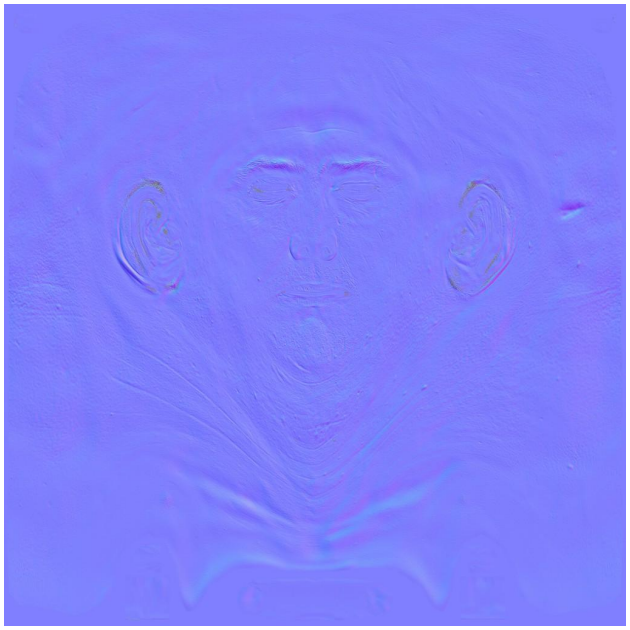
To use normal maps we need to upload the image to the GPU

Storing normals

We can either store normals in **Object space** (i.e. the *real normal value*). But we usually store them in **tangent space**



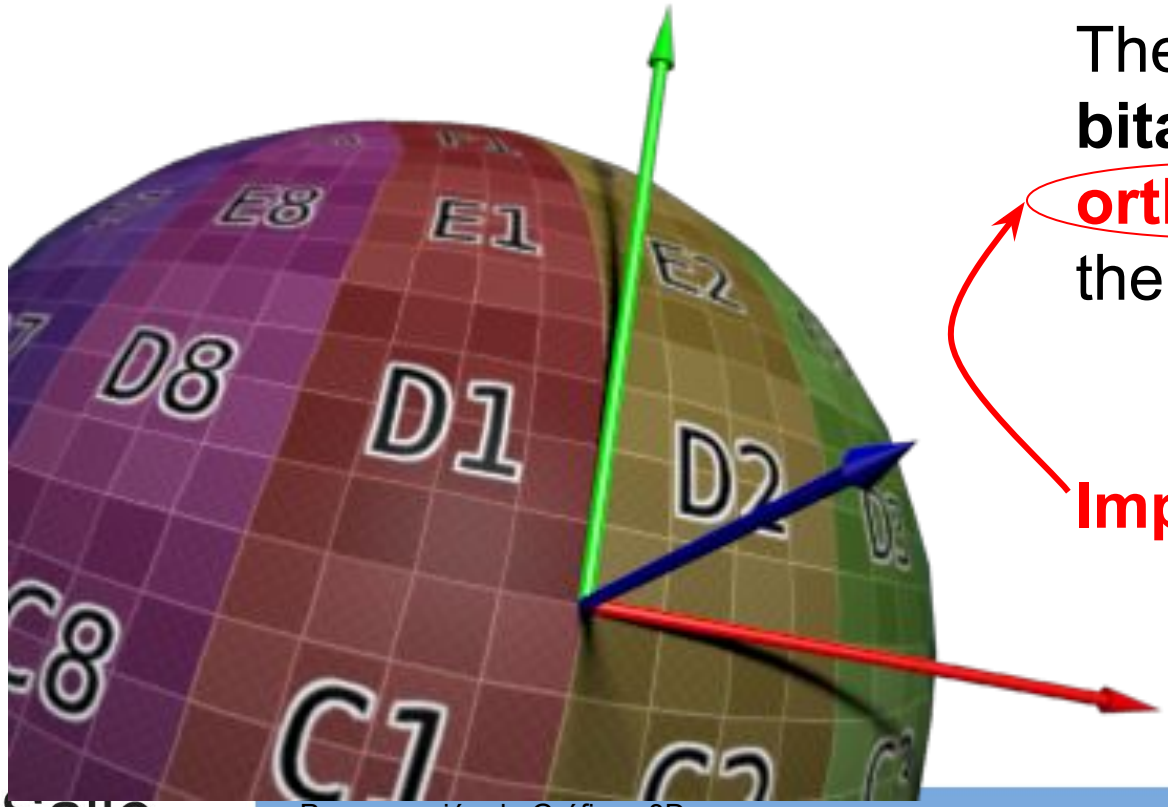
World space (“real” normals)



Tangent space (modified normals)

Tangent space

Normal value is stored *relative* to the vertex normal in **tangent space.**

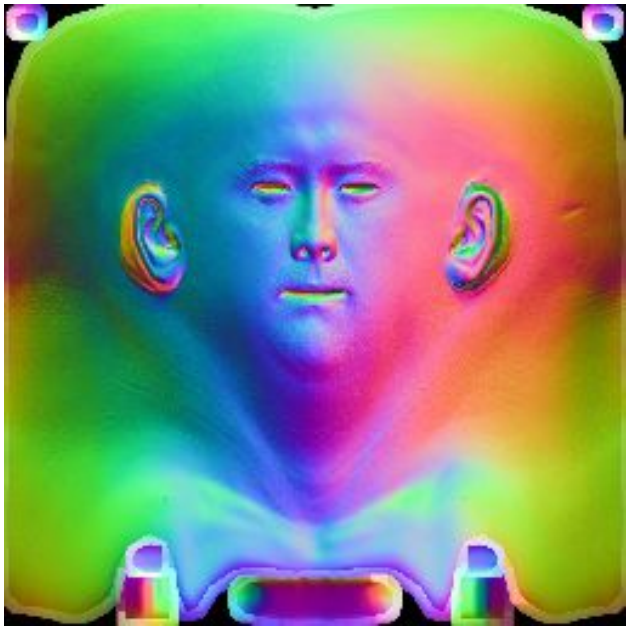


The **tangent** and **bitangent** are the **orthogonal** vectors to the normal

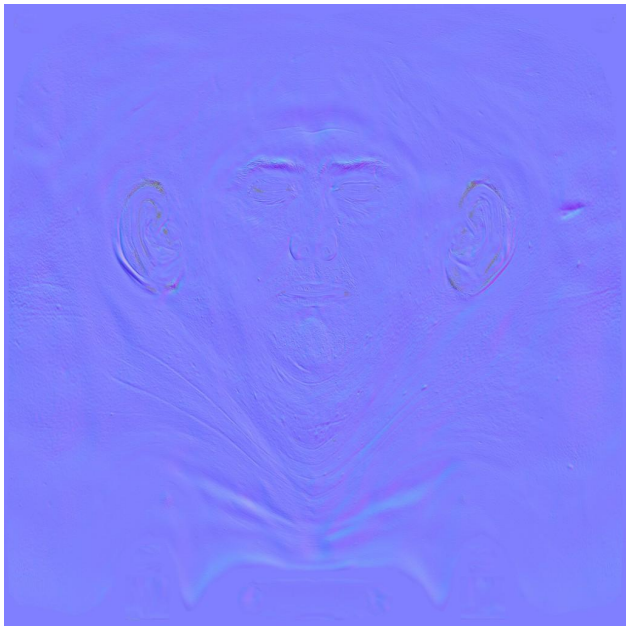
Important!!

Tangent vs object space

In this example **there is no advantage/disadvantage to using tangent/space** because the texture is not repeated



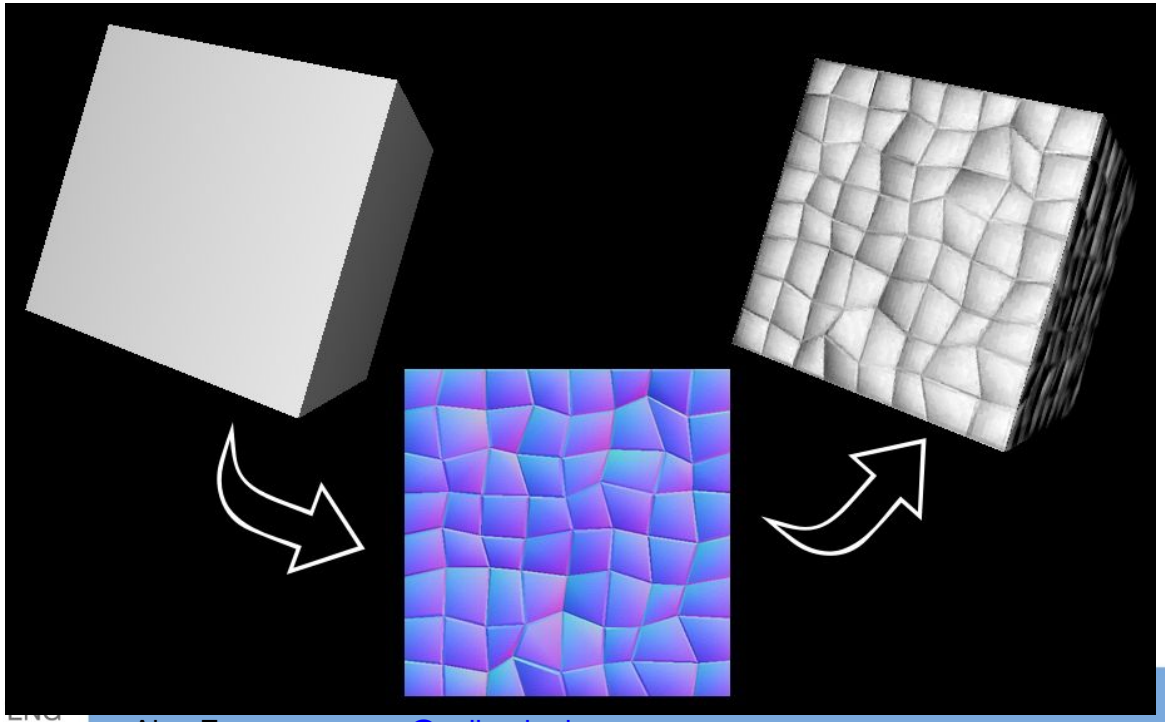
World space (“real” normals)



Tangent space (modified normals)

Tangent space vs Object space

However, in this case, tangent space is essential, because we are **reusing the same texture for multiple orientations**

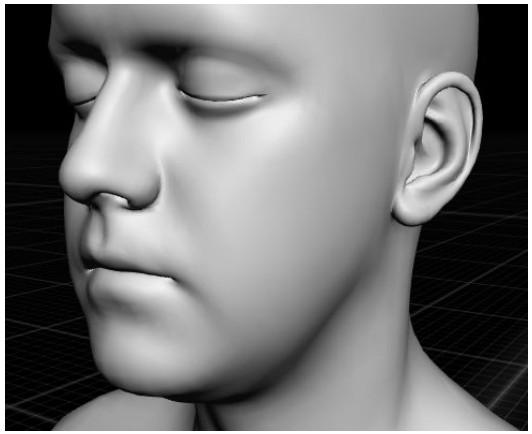


Why is tangent space blue?

The texture has a general blue tone because overall, the normal is towards the “outside of the surface”.

As usual, X is right in the plane of the texture, Y is up (again in the plane of the texture), thus given the right hand rule Z point to the “outside” of the plane of the texture.

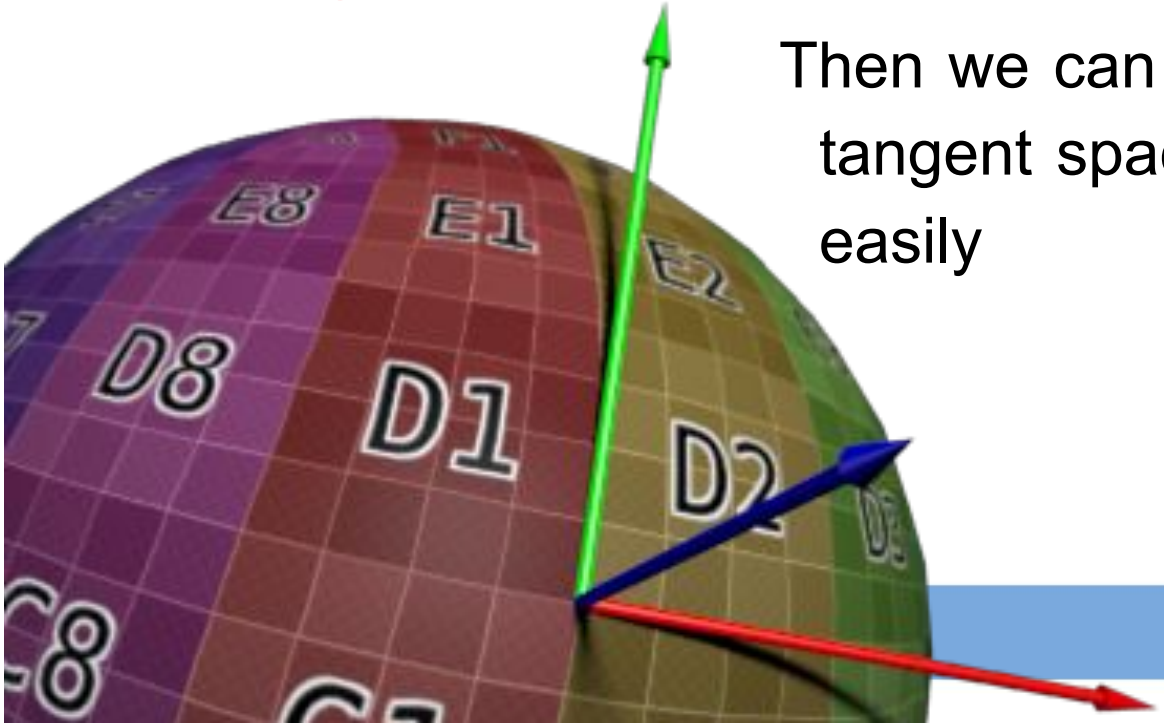
No texture / Diffuse only / Diffuse + normal



Tangent Space -> Object Space

If our normal is stored in tangent space, we need to transform it to object space before using it.

To transform for each vertex we must find the **three axes of tangent space** for each vertex.



Then we can **transform** between tangent space and object space easily

How do we transfer from one coordinate system to another?

That's right - a matrix!

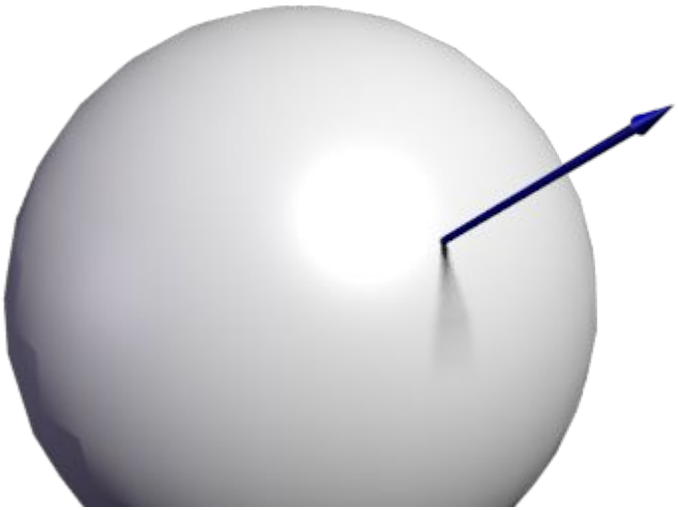
In this case, we multiply tangent space normal by the TBN matrix:

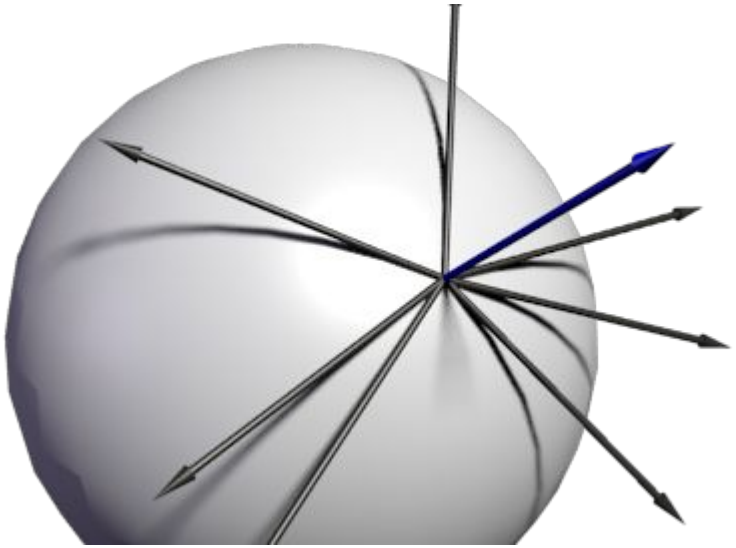
$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

T = tangent, B = bitangent, N = normal

Finding tangent space

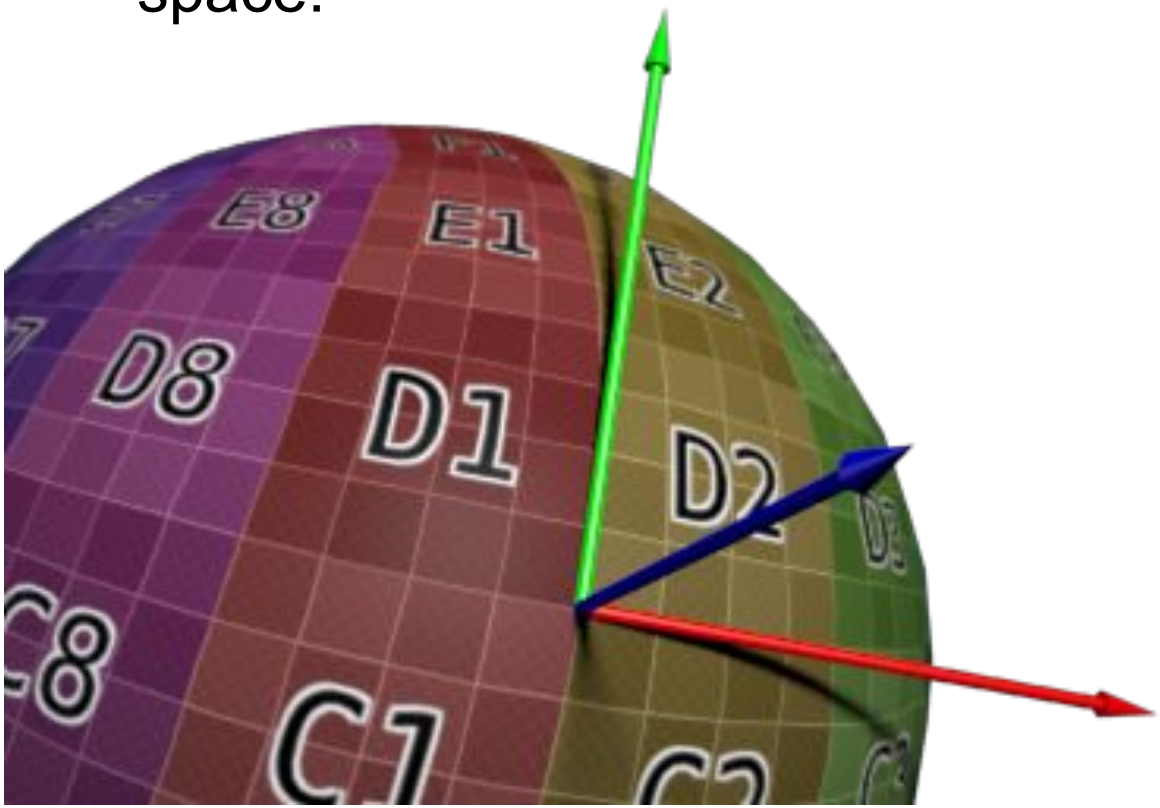
We already have a normal for each vertex. But there are an infinite number of tangents and bitangents





Finding tangent space

We use the UV mapping of the object to define tangent space:



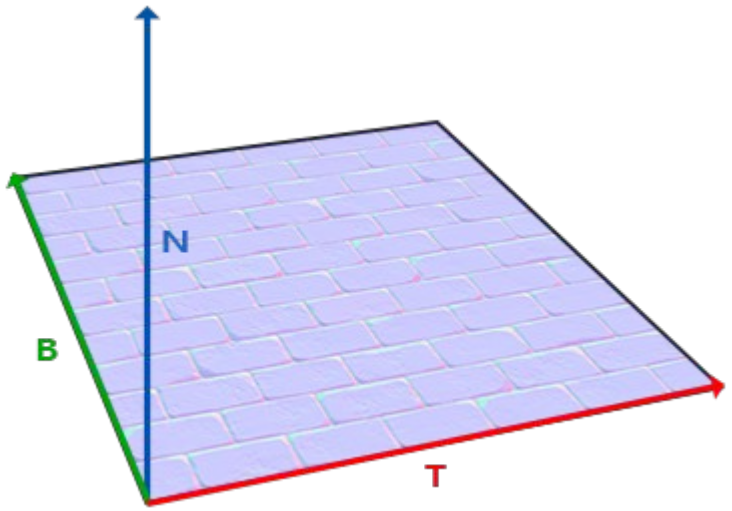
Tangent space is defined according to the **UVs of an object**

e.g. ‘up’ in tangent space is +u

‘side’ is +v

Calculating tangent space

We need to obtain the **unit tangent and bitangent vectors for each vertex**



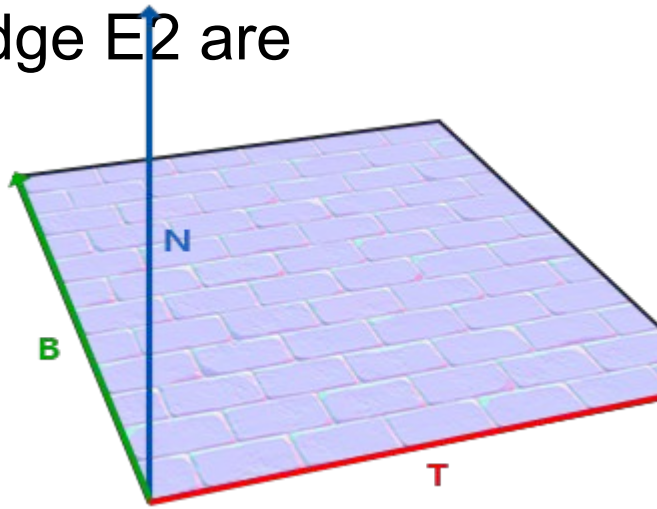
Finding uv axes

If a triangle's vertices happens to align perfectly with the UV axes, then we just take the difference in uvs along the triangle

But triangle edges rarely align with uvs!!

We need to use relative differences

Texture coordinate **differences** of an edge E2 are ΔU2 and ΔV2



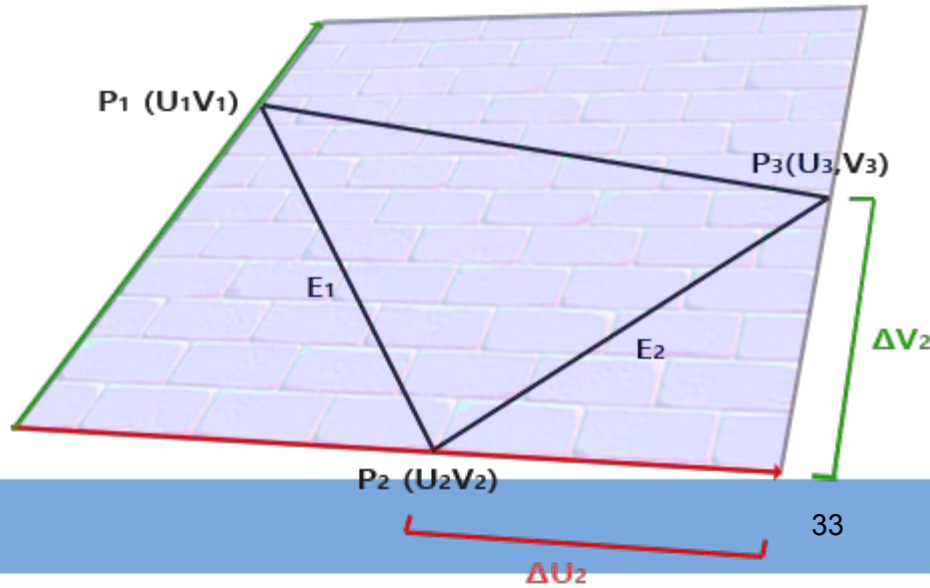
i.e

E1 = ΔU1 * T + ΔV1 * B

E2 = ΔU2 * T + ΔV2 * B

we know E1 and E2 because

we have the vertex positions



Solving the system

$E1 = \Delta U1 * T + \Delta V1 * B$

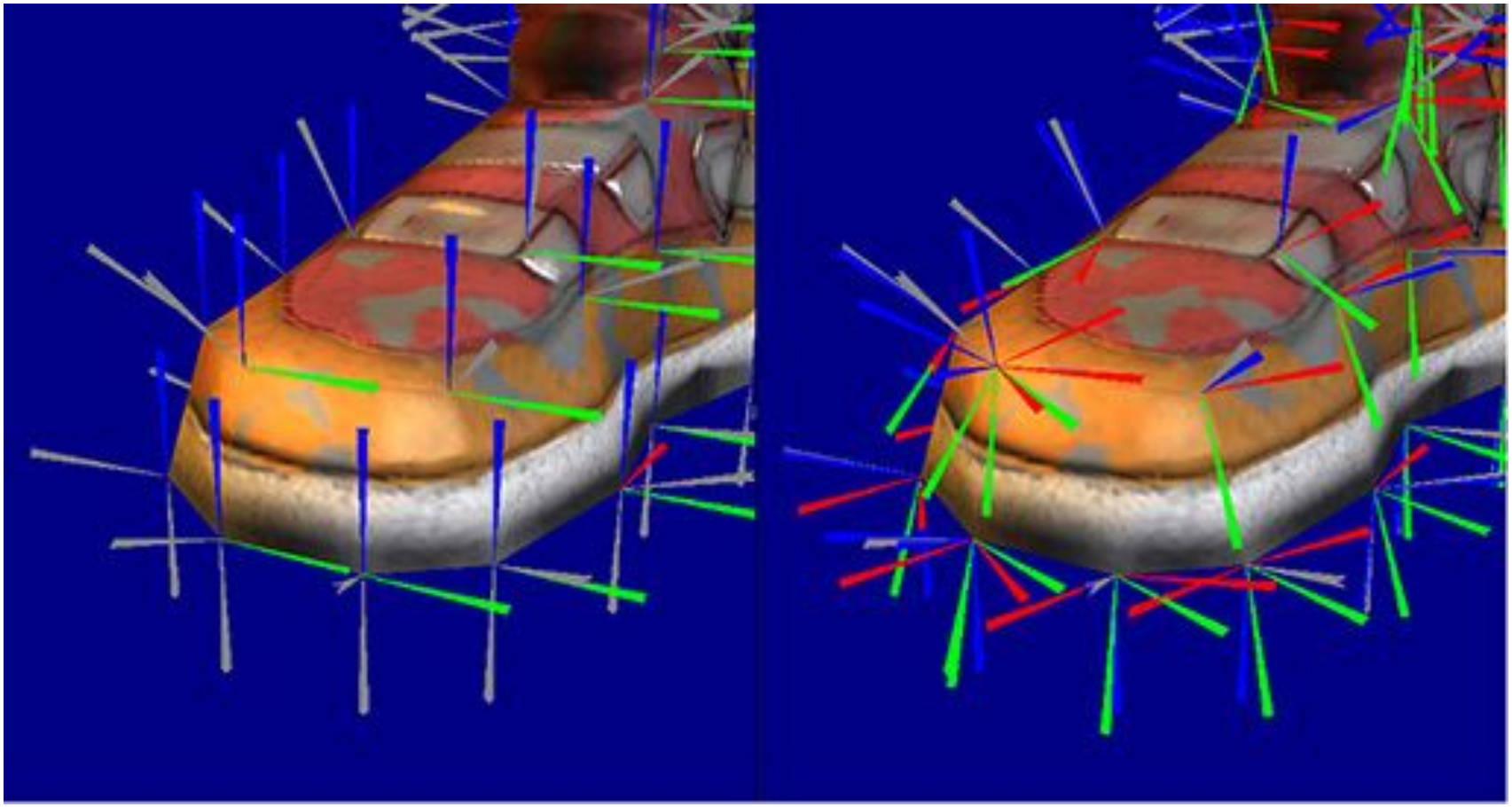
$E2 = \Delta U2 * T + \Delta V2 * B$

this linear system can be solved using linear algebra to give a single TUB matrix. This matrix allows us to transform our normals from tangent space to object space

Full derivation + code:

http://www.terathon.com/code/tangent.html

Tangent Space for all vertices of object



Solving the system

10 years ago the standard way of calculating tangent space was to **calculate the tangents and bitangents in C++**, and pass the TBN matrix to the shader.

Now, shaders have got fast enough such as that it easier to **calculate tangent space in the shader**.

(in applications with lots of geometry it is also faster, as we are passing less memory to the GPU)

derivation: http://www.thetenthplanet.de/archives/1180

Tangent space in the shader

It's only possible thanks to modern GPU design. In reality, GPUs process 'batches' of vertices and pixels. This allows us to access the 'neighbouring' vertices

the shader functions to do this are on the estudy (glsl_tangent.txt)

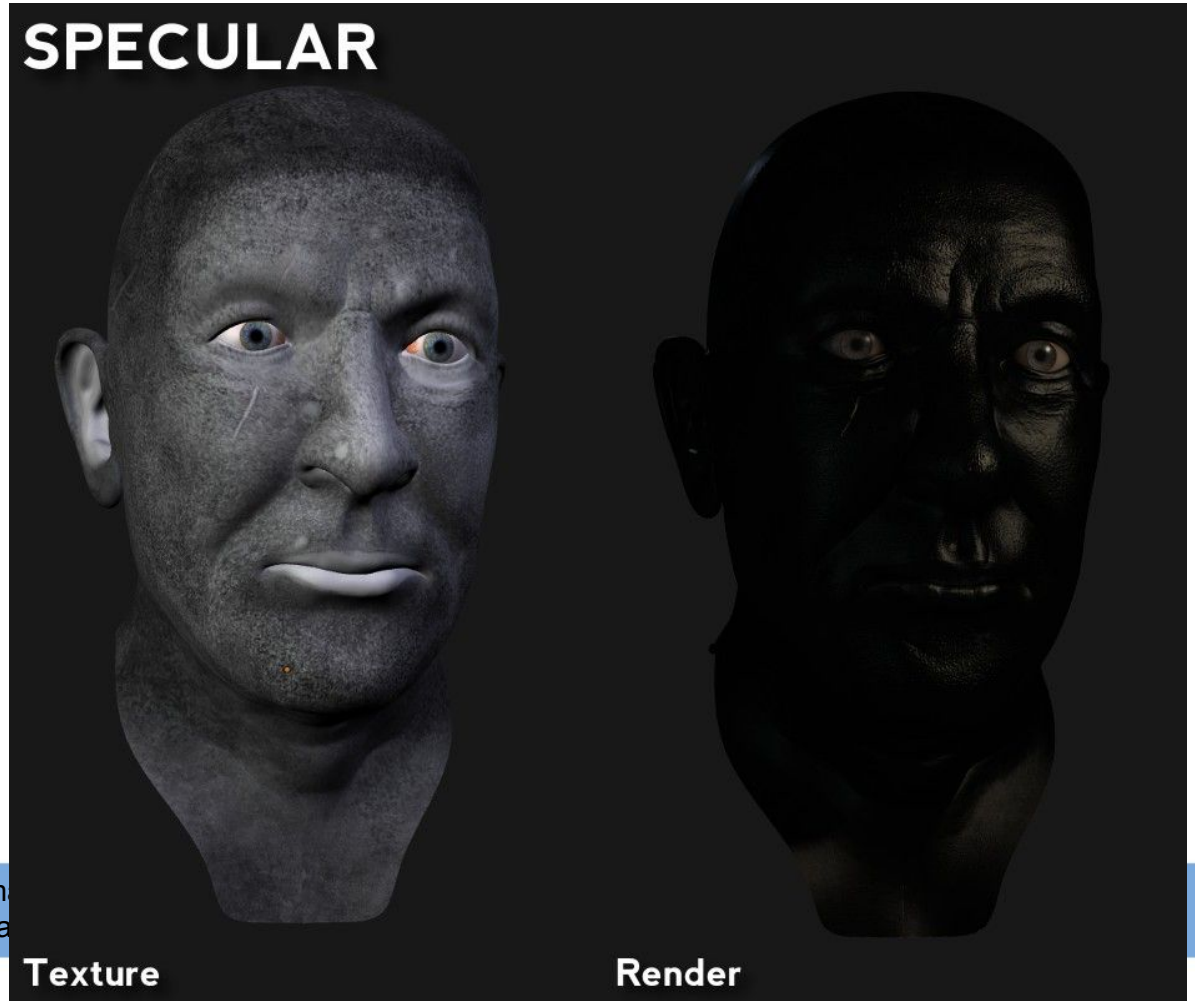
```
mat3 cotangent_frame(vec3 N, vec3 p, vec2 uv)
{
    // get edge vectors of the pixel triangle
    vec3 dp1 = dFdx( p );
    vec3 dp2 = dFdy( p );
    vec2 duv1 = dFdx( uv );
    vec2 duv2 = dFdy( uv );
    // solve the linear system
    vec3 dp2perp = cross( dp2, N );
    vec3 dp1perp = cross( N, dp1 );
    vec3 T = dp2perp * duv1.x + dp1perp * duv2.x;
    vec3 B = dp2perp * duv1.y + dp1perp * duv2.y;

    // construct a scale-invariant frame
    float invmax = inversesqrt( max( dot(T,T), dot(B,B) ) );
    return mat3( T * invmax, B * invmax, N );
}
```

```
// assume N, the interpolated vertex normal and
// V, the view vector (vertex to eye)
vec3 perturbNormal( vec3 N, vec3 V, vec2 texcoord, vec3 normal_pixel )
{
    normal_pixel = normal_pixel * 2.0 - 1.0;
    mat3 TBN = cotangent_frame(N, V, texcoord);
    return normalize(TBN * normal_pixel);
}
```

Specular textures

Specular textures are greyscale textures which allows the artist to control how much specular reflection there is on an material



Normal Specular Mapping

Add uniforms to shader.h use_normal_map etc

- WORK WITH FORWARD RENDERING for now

add ids to material

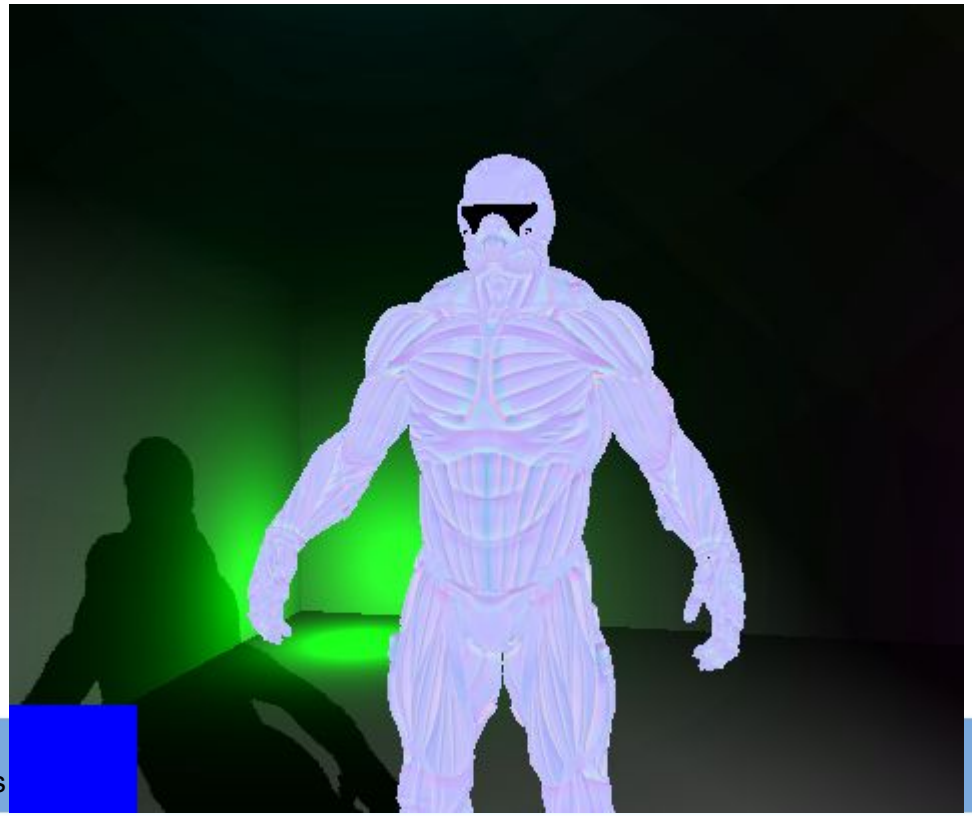
```
int diffuse_map;
int cube_map;
int normal_map; //NEW!!
int specular_map; //NEW!!
float normal_factor; //how strong are out normals
```

Upload new textures to setMaterialUniforms

```
//texture uniforms - diffuse
if (mat.diffuse_map != -1){
    shader_->setUniform(U_USE_DIFFUSE_MAP, 1);
    shader_->setTexture(U_DIFFUSE_MAP, mat.diffuse_map, 8);
} else shader_->setUniform(U_USE_DIFFUSE_MAP, 0);
//normal
if (mat.normal_map != -1) {
    shader_->setUniform(U_USE_NORMAL_MAP, 1);
    shader_->setTexture(U_NORMAL_MAP, mat.normal_map, 9);
} else shader_->setUniform(U_USE_NORMAL_MAP, 0);
//specular
if (mat.specular_map != -1) {
    shader_->setUniform(U_USE_SPECULAR_MAP, 1);
    shader_->setTexture(U_SPECULAR_MAP, mat.specular_map, 10);
} else shader_->setUniform(U_USE_SPECULAR_MAP, 0);
```

Add uniforms to shader and test

How to test? - paint textures directly

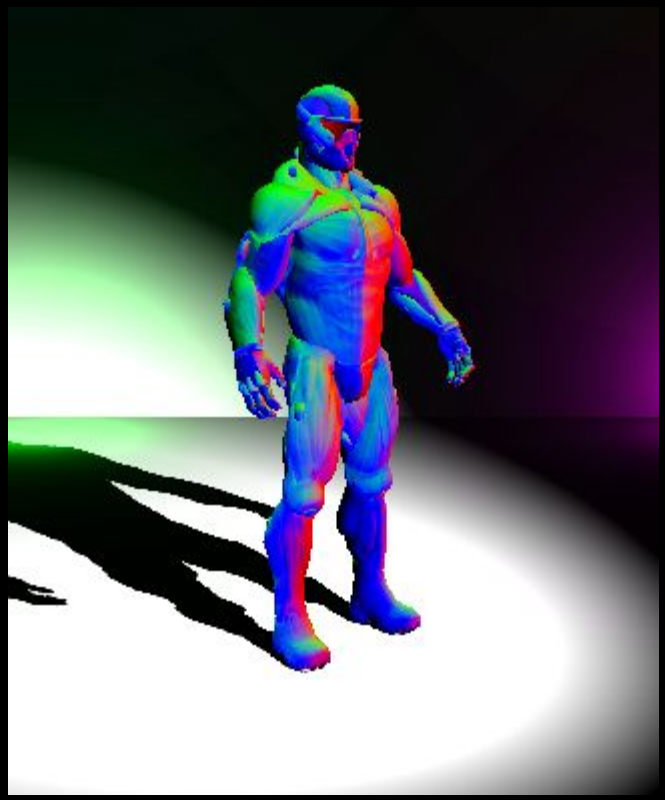


Modifying normal map

Use perturb normal function to replace normal with normal from texture

```
// perturbs the normal using a tangent space normal map
// N - normal vector from geometry
// P - vertex position
// texcoord - the current texture coordinates
// normal_sample - the sample from the normal map
vec3 perturbNormal( vec3 N, vec3 P, vec2 texcoord, vec3 normal_sample )
```

Don’t forget to only change normal if **u_use_normal_map is set to 1**!



Adding specular map

Multiply specular reflection component by color value of specular texture

again - only do it if u_use_specular_map == 1!

Task

Get normal and specular maps working!

Texture tiling

Texture tiling

Tiling is to repeat a texture multiple times over a surface.

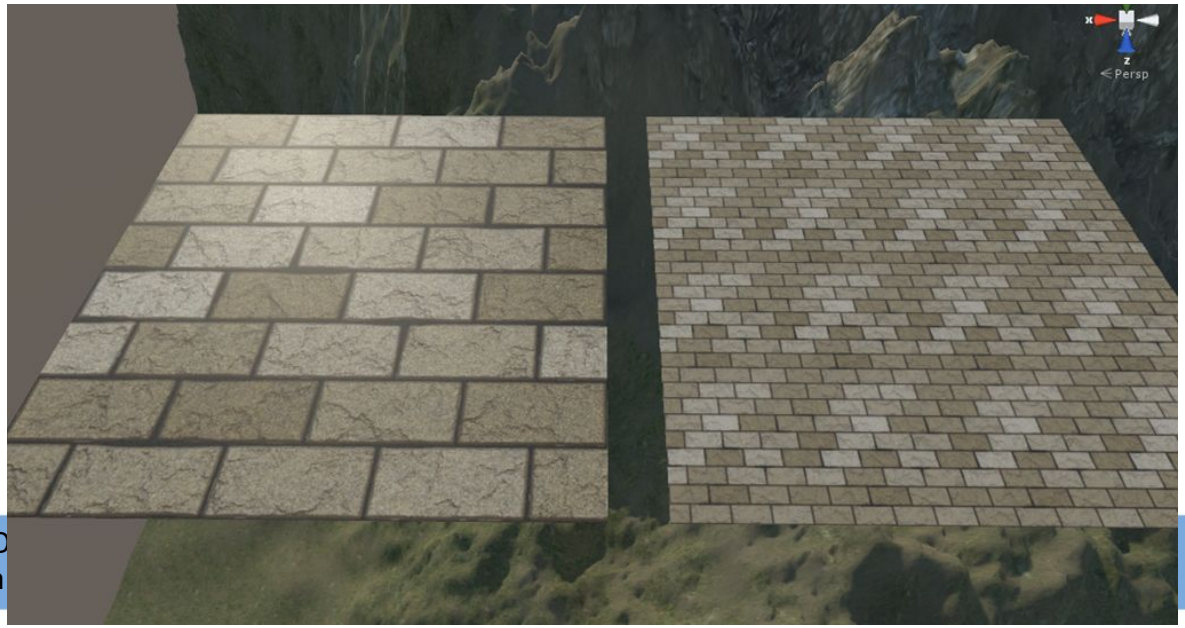
By default, UVs ‘wrap’ outside values of 0->1

i.e. a value of 3.5 = 2.5 = 1.5 = 0.5

Texture tiling

So if you scale the uvs of a geometry by a factor, you will repeat the texture

Here, the uvs coords of the plane are * 4



Texture tiling

Add uniform to shader (vec2 - because you can scale differently in u and v)

Then, in fragment shader, multiply geometry uvs by scale factor:

```
//scale uvs
vec2 s_uv = v_uv * u_uv_scale;
```

use s_uv as texture coords for *all* texture look-ups

Normal factor

The normal factor is a way of controlling the effect of normal map displacement

Task

Load the bricks_normal.tga normal map

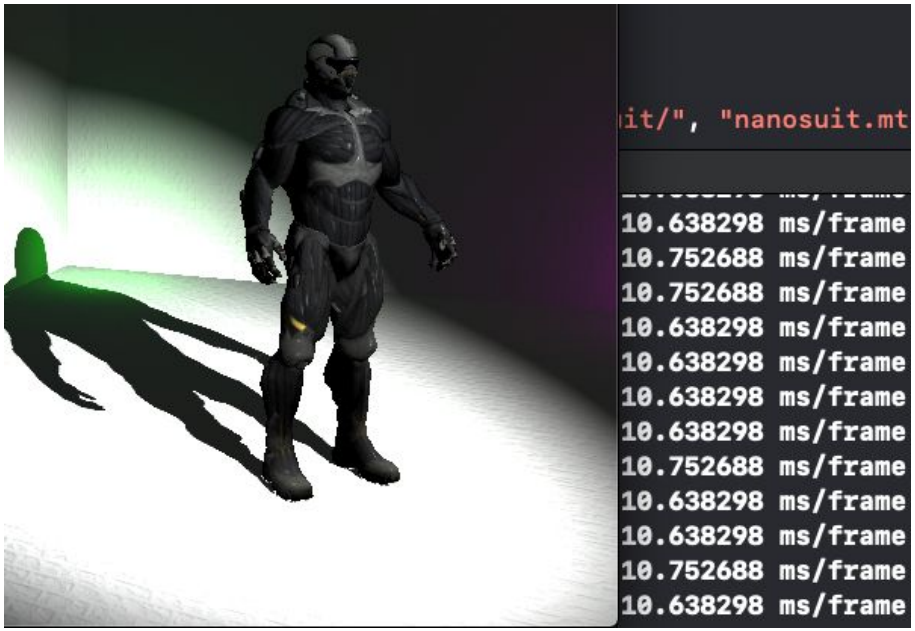
- add as normal map to material of cube

Add support for tiling

Add support for normal factor

...and finally

Task: get it all working in deferred rendering, measure performance difference.



On my machine deferred is twice as fast!!

MVD
Alun Evans – alunthomas.evans@salle.