EDAF80 Introduction to Computer Graphics

Seminar 3

Shaders

Michael Doggett

 Slides by Carl Johan Gribel, 2010-13

Today

- OpenGL Shader Language (GLSL)
- Shading theory
- Assignment 3: (you guessed it) writing shaders!

OpenGL Shading Language (GLSL)

- C like language for programming GPUs
- Vertex & fragment shaders
- Labs: **GLSL 4.1** (OpenGL 4.1)
  - Have Geometry, Tessellation shaders
- Newest: GLSL 4.6

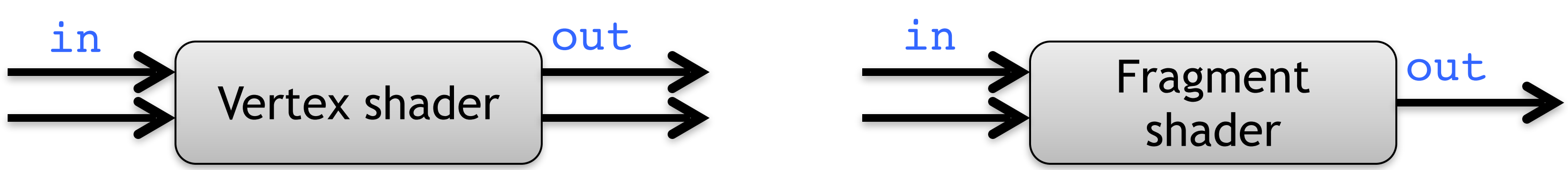
GLSL Types

- **Types**

  `float, int, bool, vec2, vec3, vec4, mat3 …`
  `sampler2D, samplerCube …`

- **Type qualifiers**

  `const` compile-time constant
  `uniform` constant per primitive
  `in, out` attributes passed to, from and
  between the vertex & pixel shader



OpenGL Shading Language

- Structures and arrays with built in operators

  `vec2, vec3, vec4, mat3, mat4`

  swizzle: `vec3 = vec4.xyz ...`

  operator overloading: `vec4 = mat4 * vec4 …`
- User defined functions
- Built-in functions

  `sin, cos, pow, normalize, min, max, clamp, reflect, refract, sqrt, noise, ...`

OpenGL Shading Language

- Flow control

  `if`, `if-else`, `for`, `while`, `do-while`

  `discard` (fragment only)
- Preprocessor directives

  `#define`, `#undef`, `#if`,

  `#else`, `#endif` …
- comments: `//`, `/*  */`

Shader development tools

- ATI RenderMonkey
- nVidia FX Composer
- Mac OpenGL Shader Builder
- GLSL Devil

  etc
- This course: by hand 🙂
  Text-editor in Visual Studio
  Compiler messages (from the GPU) written to ImGui Log window

diffuse.vert

```
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;

uniform mat4 vertex_model_to_world;
uniform mat4 normal_model_to_world;
uniform mat4 vertex_world_to_clip;

out VS_OUT {
    vec3 vertex;
    vec3 normal;
} vs_out;

void main()
{
    vs_out.vertex = vec3(vertex_model_to_world * vec4(vertex, 1.0));
    vs_out.normal = vec3(normal_model_to_world * vec4(normal, 0.0));

    gl_Position = vertex_world_to_clip * vertex_model_to_world * vec4(vertex, 1.0);
}
```

Vertex Shader Input

```
layout (location = 0) in vec3 vertex;
layout (location = 1) in vec3 normal;

uniform mat4 vertex_model_to_world;
uniform mat4 normal_model_to_world;
uniform mat4 vertex_world_to_clip;
```

- layout specifies the location of input data in the vertex buffer
- from node.cpp, Node::render. Other mat4 setup there as well
- Matrix data is loaded with `glUniformMatrix4fv(..)`

```
glUniformMatrix4fv(glGetUniformLocation(_program,
"vertex_model_to_world"), 1, GL_FALSE, glm::value_ptr(world));
```

Vertex Shader Output

```
out VS_OUT {
    vec3 vertex;
    vec3 normal;
} vs_out;

void main()
{
    vs_out.vertex = vec3(vertex_model_to_world * vec4(vertex, 1.0));
    vs_out.normal = vec3(normal_model_to_world * vec4(normal, 0.0));

    gl_Position = vertex_world_to_clip * vertex_model_to_world * vec4(vertex, 1.0);
}
```

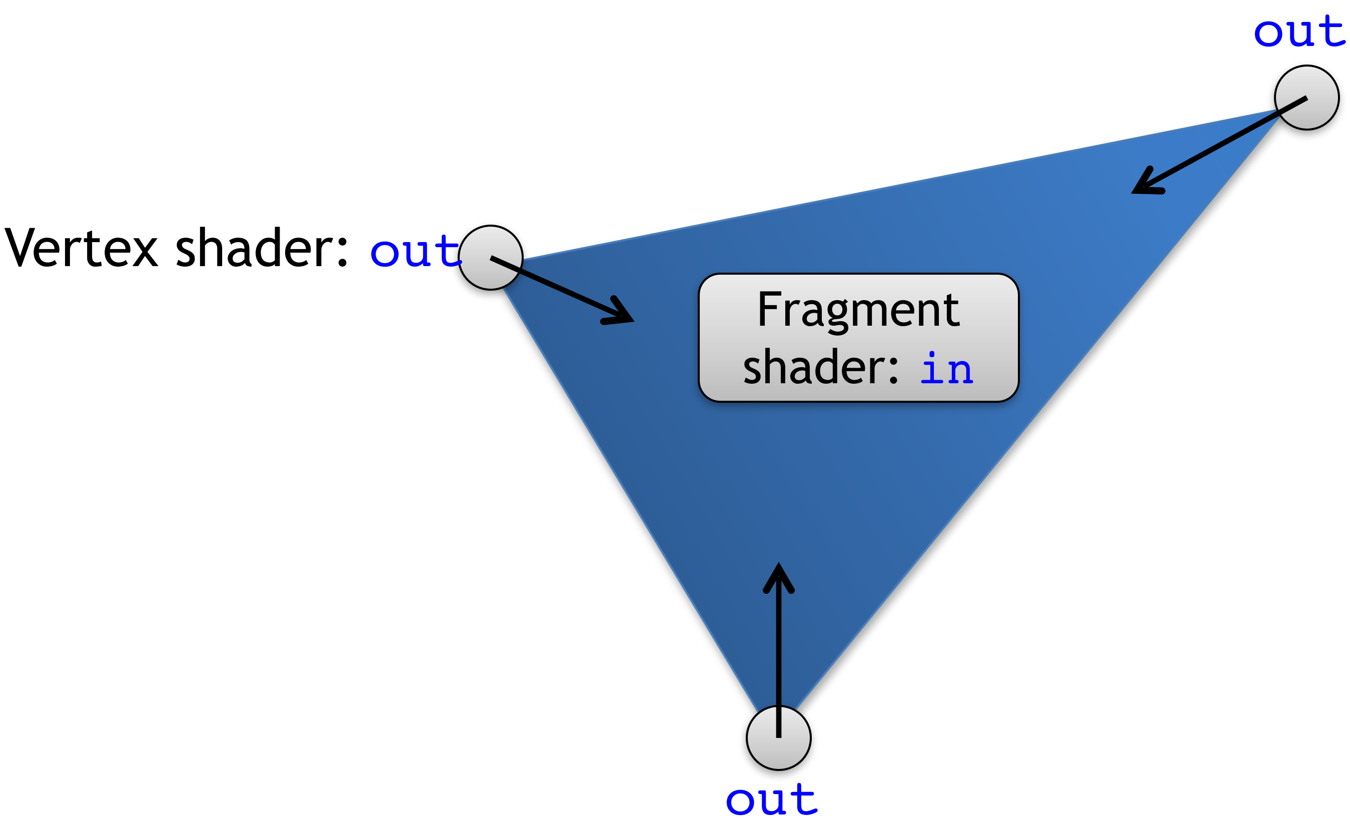
- VS_OUT is a struct with the output values
- This must match the input to the fragment shader

Fragment Shader

```
uniform vec3 light_position;

in VS_OUT {
    vec3 vertex;
    vec3 normal;
} fs_in;

out vec4 frag_color;

void main()
{
    vec3 L = normalize(light_position - fs_in.vertex);
    frag_color = vec4(1.0) * clamp(dot(normalize(fs_in.normal), L), 0.0, 1.0);
}
```

- VS_OUT struct matches Vertex Shader output and contains interpolated values
- vertex is the world position of the current pixel
- L is the normalised vector to the light
- light_position set with
```
glUniform3fv(glGetUniformLocation(program, "light_position"),
1, glm::value_ptr(light_position));
```

in/out: interpolation over triangle



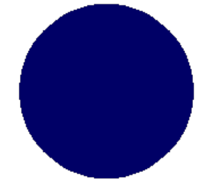
Today

- OpenGL Shader Language (GLSL) ✔
- Shading theory
- Assignment 3: (you guessed it) writing shaders!

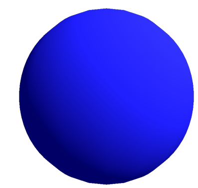
Phong shading

- Phong shading at pixel:
  (assuming normalized vectors)

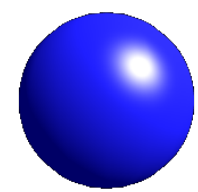
$$\text{color} = \text{AmbientColor} + \text{DiffuseColor} * \mathbf{n} \cdot \mathbf{L} + \text{SpecularColor} * (\text{reflect}(-\mathbf{L}, \mathbf{n}) \cdot \mathbf{V})^{\text{Shininess}}$$



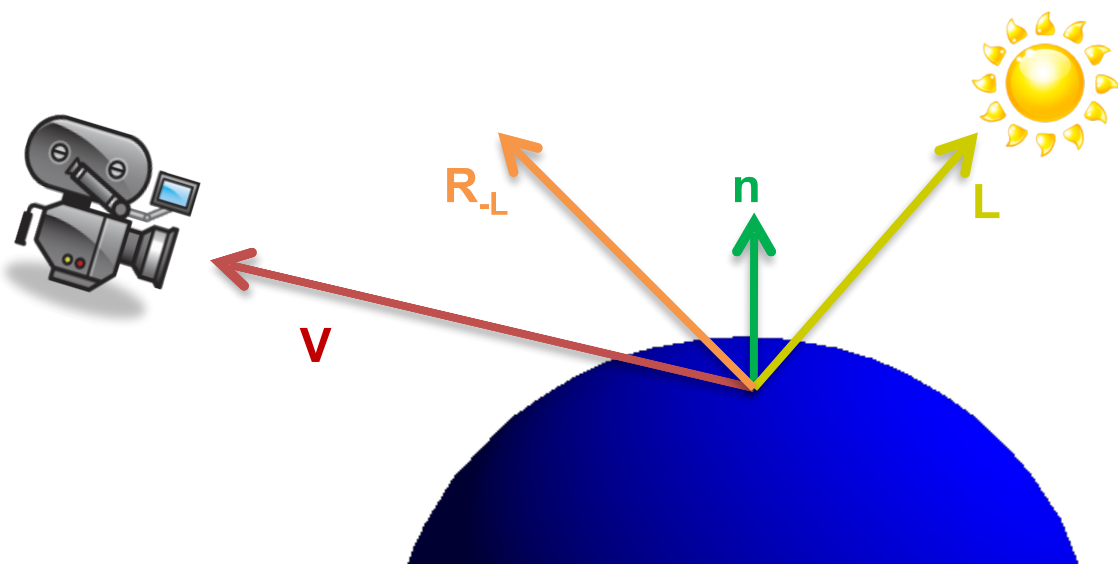
ambient



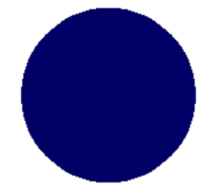
ambient + diffuse



ambient + diffuse + specular



Phong shading



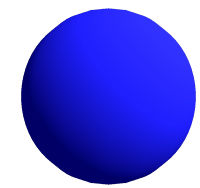
- Phong shading at pixel:

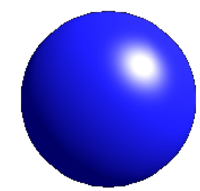
(assuming normalized vectors)

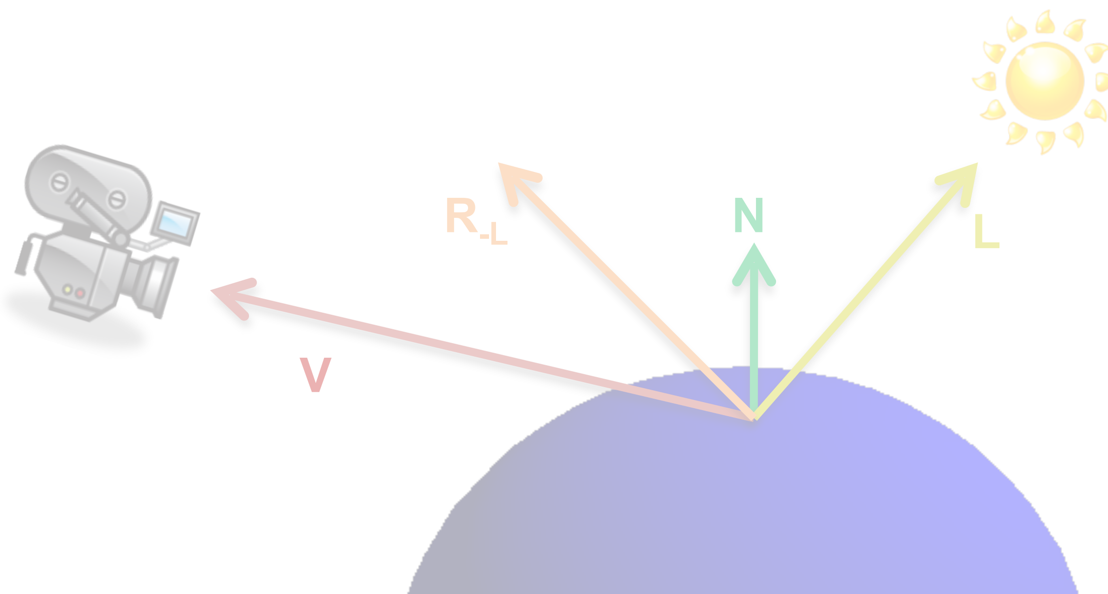
color = AmbientColor
DiffuseColor *
SpecularColor

from material, texture, cube map etc

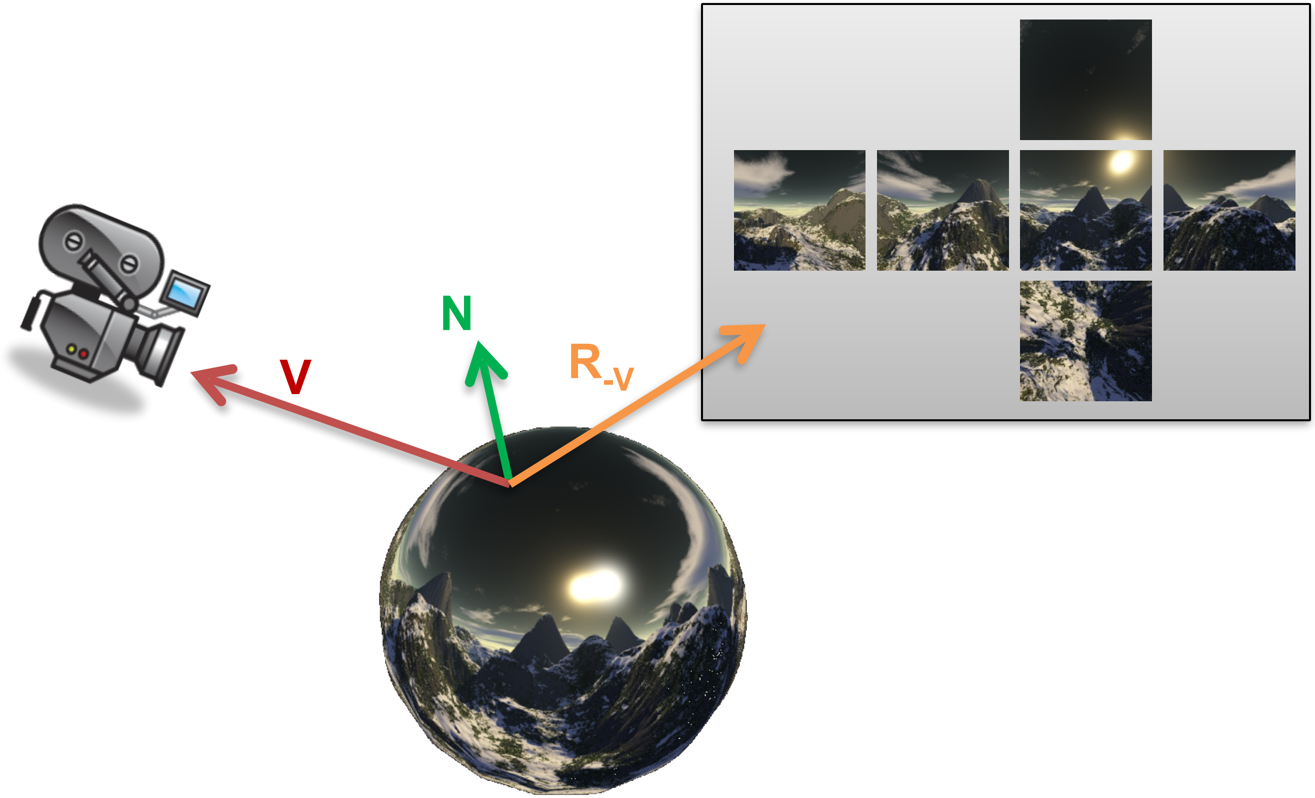
ess



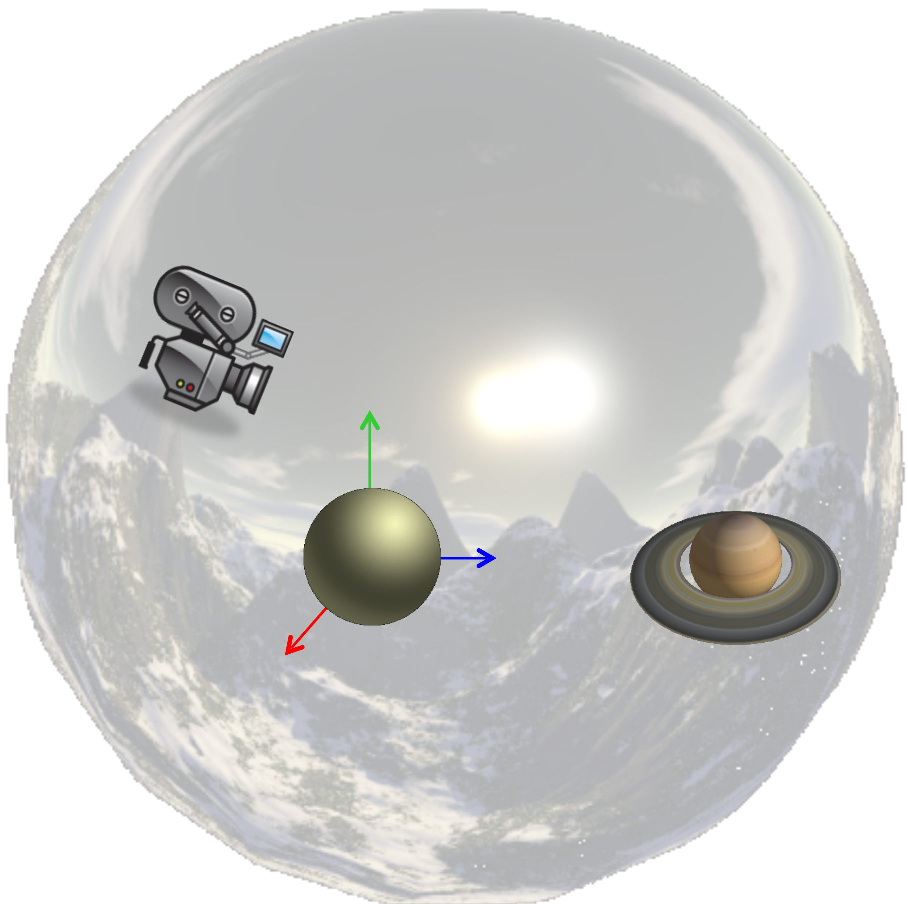




Cube mapping

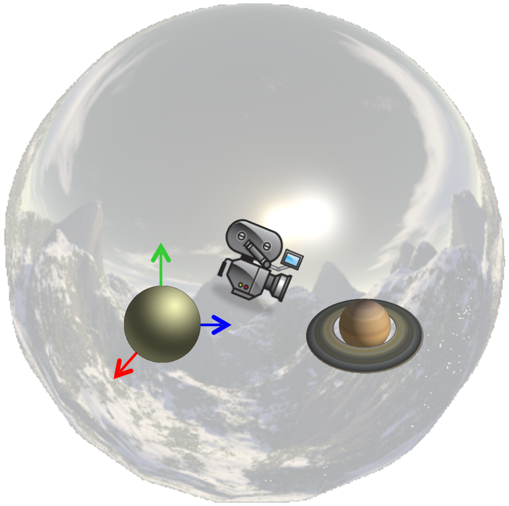


Cube mapping: Skybox



Cube mapping: Skybox

- Attach cube map to large sphere
  (large enough to contain all scene content)
- Write a “skybox”-shader that looks up colors from the cube map in the normal direction
- Make sure back-face culling is disabled
- Inside of sphere will appear as surrounding sky/landscape



Adding a Cube Map

- Complete loadTextureCubeMap and use it

```
auto my_cube_map_id = bonobo::loadTextureCubeMap(
                config::resources_path("cubemaps/Maskonaive2/posx.png"),
                config::resources_path("cubemaps/Maskonaive2/negx.png"),
                config::resources_path("cubemaps/Maskonaive2/posy.png"),
                config::resources_path("cubemaps/Maskonaive2/negy.png"),
                config::resources_path("cubemaps/Maskonaive2/posz.png"),
                config::resources_path("cubemaps/Maskonaive2/negz.png") );
```

- Add a cube map to your current node

```
big_sphere.add_texture("my_cube_map", my_cube_map_id, GL_TEXTURE_CUBE_MAP);
```

- node.cpp will bind the cube map and set the uniform (check the code)

- In your fragment shader add

```
uniform samplerCube my_cube_map;
```

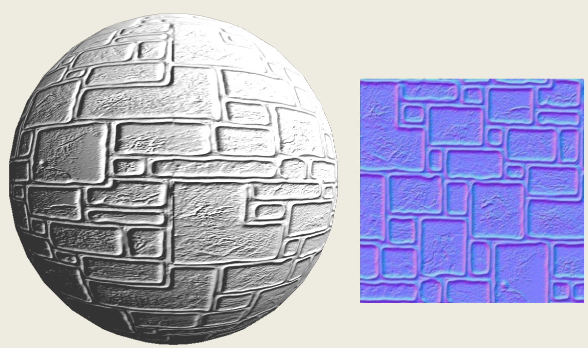
- Then use it in your fragment shader

```
frag_color = texture(my_cube_map, myTexCoords);
```

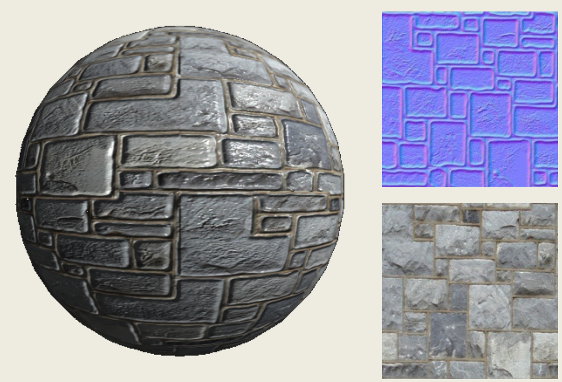
Normal mapping

- Obtain normal from normal-map instead of interpolation from vertices
- Requires a defined **tangent space**

Interpolated normal



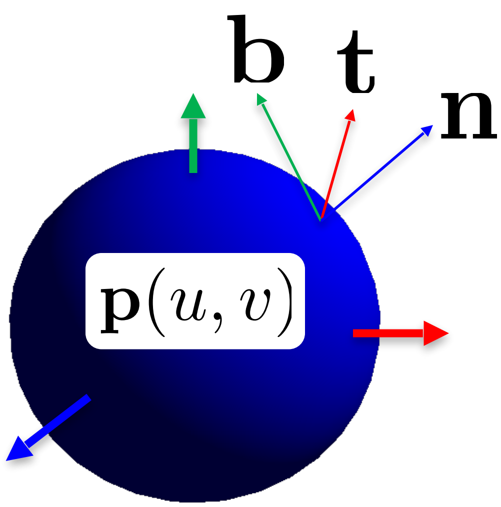
Normal from bump-map



Bump + texture

Tangent space

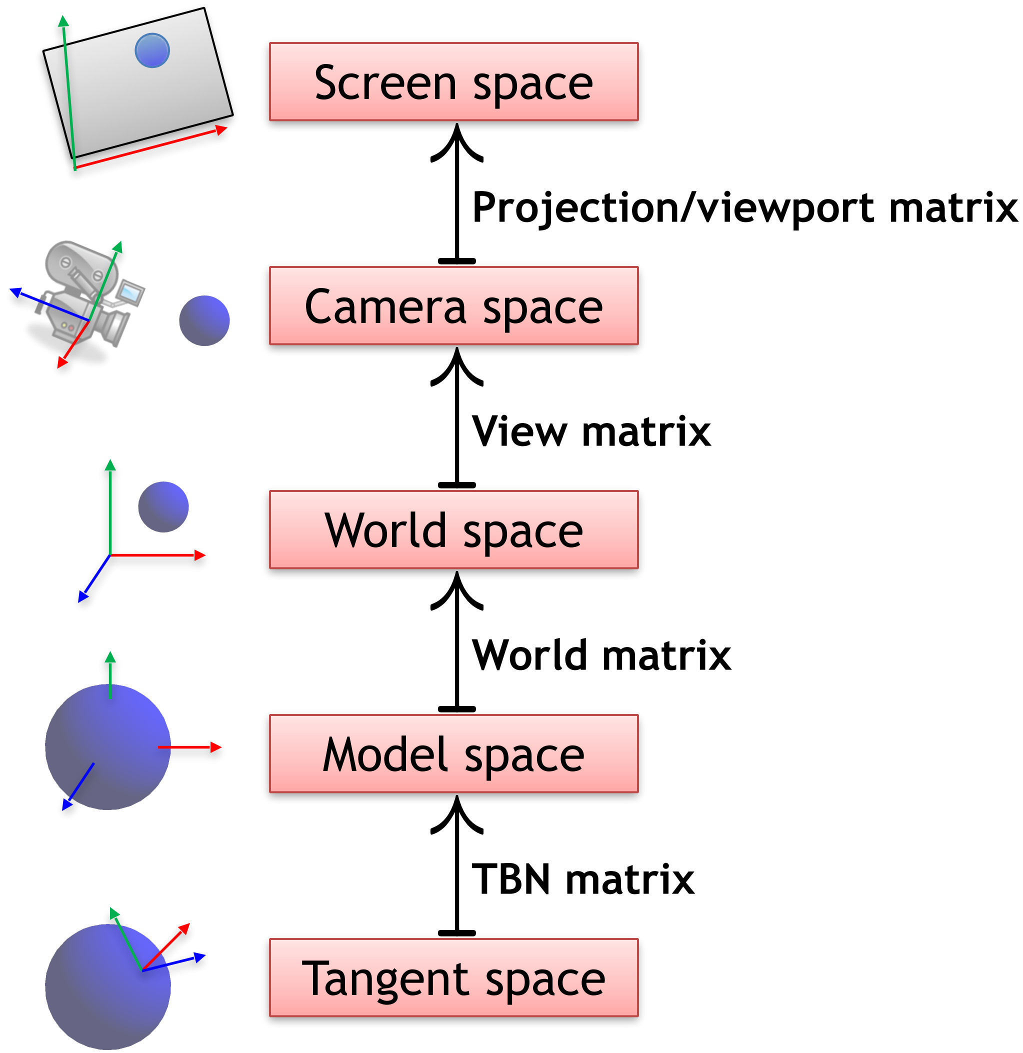
- Basis vectors: tangent **t**, binormal **b**, normal **n**
  Basis matrix: **TBN**
- Derived from the surface equation $\mathbf{p}(u,v)$
  - assignment 2!



$$\mathbf{TBN} = \begin{pmatrix} \mathbf{t}_x & \mathbf{b}_x & \mathbf{n}_x \\ \mathbf{t}_y & \mathbf{b}_y & \mathbf{n}_y \\ \mathbf{t}_z & \mathbf{b}_z & \mathbf{n}_z \end{pmatrix}$$

$$\text{tangent space} \mapsto \text{model space}$$

Spaces overview

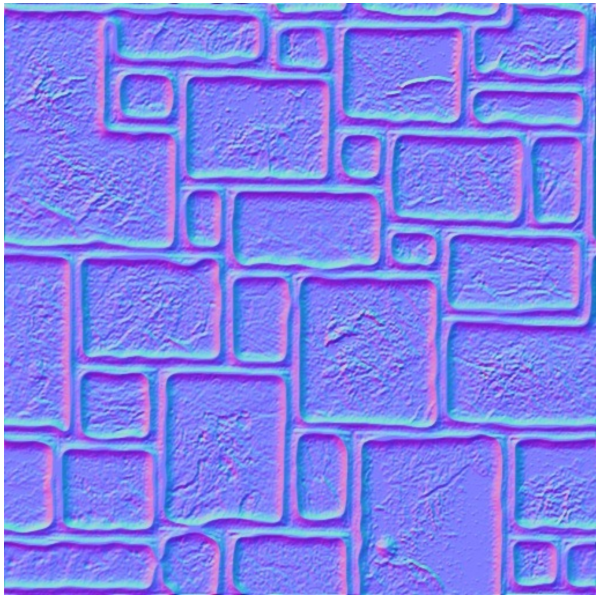


Normal map look-up

- Normals stored as (**r**, **g**, **b**) where each component range [0,1]

  Map to [-1, 1]: **n** = (**r**, **g**, **b**)*2–1

- Blue-ish color since **n** = (0, 0, 1) maps to **RGB** = (0.5, 0.5, 1)



Normal map normal

- Normal not ready to use yet; light vector is in world space, so normal must be as well

- **Transform normal from tangent to world space:**

  WorldIT * TBN * n

- Now we can proceed with light calculations

Normal map: summary

- Look up (**r**, **g**, **b**) from texture
- Map from [0, 1] to [-1, 1]: **n**
- Transform to world space: **WorldIT * TBN * n**
- Use this normal when performing light calculations

Today

- OpenGL Shader Language (GLSL) ✔
- Shading theory ✔
- Assignment 3: (you guessed it) writing shaders!

Assignment 3

- Implement the following shader techniques:
  Phong shading
  Cube mapping using a Skybox
  Normal mapping
- Use one ShaderProgram (vs + fs) per technique

- Use ‘R’ key to reload shaders

Normal mapping shader

- All tangent space basis vectors are needed (normal, tangent, binormal)
- Complete vertex definiton:

```
struct Vertex
{
	f32	x, y, z,               /* vertex position */
		texx, texy, texz,      /* texture coord's */
		nx, ny, nz             /* normal */
		tx, ty, tz             /* tangent */
		bx, by, bz;            /* binormal */
};
```

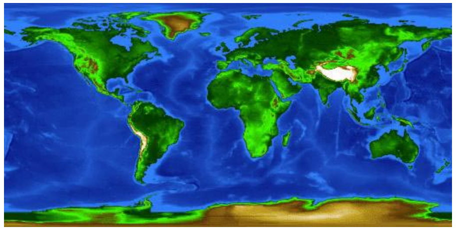
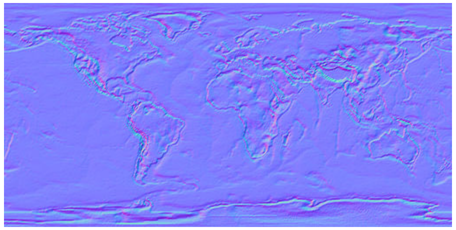
Normal mapping shader

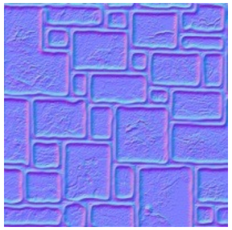
- **Hint**

  Set **t**, **b**, **n** as `in`-attributes to the vertex shader and make them vary per fragment using `out`

  Construct **TBN**-matrix in fragment shader using `mat3`

- Check out `res/textures` for more textures



Today

- OpenGL Shader Language (GLSL) ✔
- Shading theory ✔
- Assignment 3: writing shaders! ✔
- More questions about code/OpenGL/GLSL?
  - Ask on the Canvas discussion page
  - learnopengl.com