# Foundations of 3D Computer Graphics
# Material System

## Assignment Objectives

In this assignment we will also upgrade the scene graph infrastructure to allow for multiple types of materials with different GLSL shaders in a single scene. One particular material that you will help implement is a bump mapping material.

## Task 1: Material Infrastructure

So far we have been using only one GLSL shader per frame during our rendering, as controlled by the global variable `g_activeShader`, an index into the global array of `ShaderState`'s `g_shaderStates[]`. A more complex graphics program, such as a modern game, will use quite a few more to model different surface properties. In this assignment, we will incorporate a material infrastructure that allows you to use multiple shaders in a single frame. The following sections explain the new classes that will be included in your program. Note that not until the last sub section should you start altering your codes.

### `Uniforms`

Conceptually, each shader takes in a bunch of uniform variables. Some of these uniform variables, such as the projection matrix, hold the same value for all shaders, and are set once at the beginning of the frame by you. Originally, it was set directly to the currently active GLSL shader. But if we start using a different GLSL shader in the same frame, we will need to set the values of these uniforms again, because different GLSL shaders do not know about the values of each other's uniform variables . This suggests we need some kind of data structure to hold the values of these uniform variables. Then when a new shader is selected, we populate its uniform variables with values from this repository.

In the supplied code, the `Uniforms` class defined in `uniforms.h` serves this purpose. It is essentially a dictionary mapping string name to uniform values. Too use it , you would write

```
// Suppose uniforms is of type Uniforms, and m is of type Matrix4
uniforms.put("uProjection", m);

// Suppose light is of type Cvec3
uniforms.put("uLight", light);

// Set uColor variable to red
uniforms.put("uColor", Cvec3(1, 0, 0));

// You can even chain the put, since put returns the object itself
uniforms.put("a", 1)
        .put("b", 10)
        .put("c", Cvec2(1, 2));
```

As you can see the first argument of `Uniforms::put` is the name of uniform variable as you have declared in the shader code. The second argument is the actual value. Right now `put` is overloaded to take the following types as the second parameter:

- A single `float`, `int`, or `Matrix4`.
- A `Cvec<T, n>` with `T` being `float`, `int` or `double`, and `n` being 1, 2, 3, or 4. In plain English, a `int`, `float`, or `double` `Cvec` with size 1, 2, 3, or 4.
- A `shared_ptr<Texture>` type.

The last bullet needs further explanation: If you recall from the "Hello World 2D"project, shaders can access textures by declaring a `sampler2D` typed uniform variables, e.g.,

```
uniform sampler2D uTexUnit0;
```

It can then use the GLSL function `texture()` to look up values in the texture. However, the value that you supply to this uniform variable is an integer telling it which texture unit to use, as opposed to the actual texture handle. So before calling the shader, you need to manually bind the texture to the right texture unit first by calling the right sequence of `glActiveTexture` and `glBindTexture`.

In the provided material framework, the binding of texture to texture units is automatically done behind the scene. So to provide the value of a `sampler2D` variable in a `Uniforms` object, you simply put in a shared pointer to the actual texture, encapsulated by the `Texture` class (more on this later).

Now to recap, in the original codes, you set any uniforms directly by pulling the handle from `ShaderState`, and then calling one of `glUniform*` variants. In the new code, you will set the value to a `Uniforms` class, which will hold on to the value until it is needed by a shader. Thus, for the shape nodes `SgShapeNode` in the scene graph, its `draw` method now takes in a `Uniforms` object as input, instead of a `ShaderState` object. Similarly, the constructors for the `Drawer` visitor and the `Picker` visitor take in a `Uniforms` as argument, as opposed to a `ShaderState`.

### RenderStates

Recall that when you draw the arcball, you enclose it with a pair of `glPolygonMode` calls to make it drawn in wireframe. OpenGL has a "state-machine" model where after you have changed some states, whatever drawing commands following it will using the new states. Hence after you have instructed OpenGL to draw things in wireframe, and drawn the arcball, you need to turn off the wireframe with another `glPolygonMode` call. This will quickly become hard to manage since different material properties might need different OpenGL state flags to be set, and your code will quickly become peppered with these state changing calls. For example, in the "Fur" assignment, you will need to set states that controls how transparent objects are drawn. One of the state you will need to change for that assignment, is to enable *framebuffer blending*, by calling `glEnable(GL_BLEND)` before drawing the transparent objects. Afterward, you will need to call `glDisable(GL_BLEND)` to disable blending for non-transparent objects.

To make your job easier, we have introduced a new class called `RenderStates`. It stores a subset of OpenGL states (that set will become larger and large as more states become relevant for our assignment). For example, it contains a `polygonMode` member function which takes the same arguments as `glPolygonMode`. The difference is that, when you set a state using `RenderStates`'s member function, that state does not immediately take effect in OpenGL, but is stored within the object. Later, when you want a `RenderStates` internal states to take effect in OpenGL, you will call its `apply()` member function. Even when you do not call a state changing member function such as `polygonMode`, the `RenderStates` has a default value for that particular states. Thus, each `RenderStates` stores the values of the same set of OpenGL states. After a `RenderStates` has been `apply`'ed, the current OpenGL states will agree with the `RenderStates`'s set of states.

So why would this be useful? Consider the following code snippets

```
// All three  have a default polygonMode set to GL_FILL, and blending disabled
RenderStates r1, r2, r3;

// set r2 to be used for wireframe rendering
r2.polygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

```
// set r3 to be used for transparent objects
r3.enable(GL_BLEND);

// At this point, actual OpenGL states have not been changed yet.

// Now we can switch between the three sets of render states easiliy

r2.apply(); // after this, GL states correspond to that of r2

// draw stuff in wireframe, and translucent;

r1.apply(); // after this, GL states correspond to that of r1

// draw stuff not in wireframe, and non translucent

r3.apply(); // after this, GL states correspond to that of r3

// draw stuff not in wireframe and translucent.
```

As you can see, there is no saving and restoring of OpenGL states. All of that are handled by the `RenderStates` internal implementation.

### `Geometry` and `Texture`

You have already encountered the `Geometry` and `Texture` classes before. `Geometry` encapsulates the OpenGL VBO and IBOs handles, whereas `Texture` encapsulates the OpenGL texture handles. So far we have had one type of `Geometry`: static geometry with a single vertex layout as defined by `Vertex`. Similarly we have had a single type of `Texture`: 2D image maps. A more complex graphics program will often feature more different types of geometries and textures. For example: Some geometry might need to be generated dynamically and rendered on the fly; Cube texture maps (which are six 2D image textures that are mapped on an imaginary cube) might be needed to model reflection effect; and so on.

Given that we are introducing a new Material system, and both geometry and texture need to interact with it, it is now a good time to refine our `Geometry` and `Texture` definitions a bit so that they play well with the new Material system, and can be extended to support different kind of geometries and textures in the future (definitely in the next assignment). The new `Geometry` and `Texture` classes are both abstract classes with certain functions that once implemented, allow custom Geometry and Texture type to be used by the Material system. You should look at the detailed comments accompanying their definition in `geometry.h` and `texture.h` for details.

To demonstrate how custom types of geometry and textures, we supplied three concrete implementation of `Geometry` and `Texture`, in `geometry.h` and `texture.h`.

- `GeometryPN`: A `Geometry` that provides two vertex attribute streams, Position and Normal.
- `GeometryPNTBX`: A `Geometry` that provides five vertex attribute streams, Position, Normal, Tangent, Binormal, and teXture Coordinates.
- `ImageTexture`: A `Texture` that loads a PPM file with three channels, and optionally store the content in SRGB color space.

### `Material`: Pulling all together

Given the above, the actual `Material` class is a light-weight container of the following:

- It keeps a shared pointer to the actual GLSL shader program used.
- It keeps a `Uniforms` field, accessible through its `getUniforms()` member function.

- It keeps a `RenderStates` field, accessible through its `getRenderStates()` member function.

The `Material` has a `draw(geometry, extraUniforms)` member function that takes in a geometry, and some extra uniforms. This function will first set the GLSL shader, and set all the `RenderStates` it contained. Next it binds all uniforms used by the shader, first by looking at its own `Uniforms`, and then looking at the passed in `extraUniforms`. Textures are properly bound and set at this stage as well. Then it makes sure that the supplied `geometry` can provided the vertex attributes used by the shader, and enables the corresponding vertex attribute arrays. It then call `geometry`'s `draw` function to draw the geometry. Finally it disables the previously enabled vertex attribute arrays.

As an example, suppose you want to use the new Material system to draw the arcball as a wireframe sphere with solid color, you would do the following

```
// Assume uniforms is of type Uniforms and already contains the projection matrix
// Assume g_arcballMat corresponds to a Material with the right renderstates and uniforms set

// Calculate the arcball's MVM

// Record it in uniforms
sendModelViewNormalMatrix(uniforms, MVM, normalMatrix(MVM));

// Draw with the material, assume g_sphere points to a Geometry for a sphere
g_arcballMat->draw(*g_sphere, uniforms);
```

The constructor of `Material` takes in two arguments, the file name of the vertex shader, and the file name of the fragment shader. And to make your life easier, if the file names contain the substring "-gl3" and the global flag `gl_Gl2Compatible` is set, the corresponding shader files with "-gl2" substring will be loaded. So in your code, you only need to supply one set of file names, the ones ending with "-gl3". You also easily copy the Material using assignment operation.

## Plug into the Scene Graph

Our last step is to plug the material infrastructure into the scene graph. Also mentioned before, a shape node's `draw` function now takes in a `Uniforms` instead of `ShaderStates` as input. The same goes for the `Drawer` and `Picker` visitors.

Then our concrete implementation of the shape node, `SgGeometryShapeNode` is a simple wrapper around a shared pointer to a `Material`, a `Geometry` and the affine matrix. Its constructor now takes in a shared pointer to a full `Material`, as opposed to a simple color. Different `SgGeometryShapeNode`s in the scene graph can use completely different materials. Its draw method will use the `Material` to draw the `Geometry` contained in the same node.

There is one catch: when we're picking, we want to use one `Material` for all shapes. To accomplish this, we keep a global variable called `g_overridingMaterial` which is a `shared_ptr<Material>`. If it is `NULL`, the `draw` function of `SgGeometryShapeNode` will do its work using its own material as usual. If `g_overridingMaterial` is not `NULL`, whatever it points to will be used by `SgGeometryShapeNode::draw()` as the material.

## Finally, Code Migration

To start off, **make a copy** of your current project directory just in case you need to go back to it.

Now copy all the files in the starter to your project directory, replacing files with same names in your current directory. Note that you have helped finishing the codes in `picker.cpp`, and `scenegraph.cpp` in the "Hierarchical Transformations" assignment. So you can either choose to copy the code you have written to the new files, or just use the newly provided files. Since what you wrote implements a well specified method, your code and the solution code should accomplish the same thing, and using either version should be fine. The `shaders` directory should also replace your project's existing `shaders` directory (Actually there're only two new shaders, `normal-*.fshader` and `normal-*.vshader`).

If you are on Mac or Linux, the new `Makefile` is already included, and you should change your main program file to `asst6.cpp`. If you are on Windows, add all the `cpp` and `h` files to your project, **except** `asst6-snippets.cpp`.

Finally follow the instructions in `asst6-snippets.cpp` to change your `asst6.cpp` to use the new Material system. As you run the resulting program, you will notice that

- The robots are drawn with diffuse color
- The arcball is drawn with wireframe and solid color
- The ground is drawn with a texture.

# Task 2: Bump Mapping

To make your pictures look even snazzier, you will help finishing our bump mapping implementation.

## Make Light Movable

Your first goal is to make the two lights of the scene pickable and movable. This is a very low hanging fruit given our scene graph structure, and will help your inspect the bump mapping effect once you finish it.

Instead of storing the world position of the light as two `Cvec3`'s `g_light1` and `g_light2`, you want to store them as Transform nodes in the scene graph. The origin of these frames serve as the position of the lights, and you can attach a Shape node to the Transform nodes so that the position of the light become visible, pickable, and manipulatable (e.g., just attach a sphere)

To pass the lights' eye space coordinates to the shader, you will need to do something like the following:

```
// get world space coordinates of the light
Cvec3 light1 = getPathAccumRbt(g_world, g_light1Node).getTranslation();

// transform to eye space, and set to uLight uniform
uniforms.put("uLight", Cvec3(invEyeRbt * Cvec4(light1, 1)));
```

assuming your `g_light1Node` points to the `SgRbtNode` corresponding to the first light.

Refer to how `g_groundNode` is set and added to `g_world` in `initScene()` on how to do this. (Don't forget to call `g_world`'s `addChild`!) You probably want to use `g_lightMat` as the material for the shape node.

## Write Some GLSL

First of all, we will give you a texture `FieldstoneNormal.ppm`, but the 3 values making up its pixel will not be interpreted as RGB values, but rather as the 3 coordinates of a normal. If you refer to `initMaterials()` that you added inserted to your program, you will see that this texture is bound to the `uTexNormal` uniform variable of the `g_floorMat` material, which uses the shaders `normal-gl{2|3}.{f|v}shader`.

You need to change the fragment shader `normal-gl{2|3}.fshader` (depending on your `g_Gl2Compatible` setting) to read the appropriate pixel from the texture, transform it to eye space, and use it for shading calculation. This will give your geometry a high resolution look.

**Texture Coordinates**: each vertex will need $(x, y)$ texture coordinates. We will build these in to our ground, cube, and sphere geometry objects. These attributes are passed into the vertex shaders for the draw call. You can access them as `vTexCoord` in the fragment shader.

**Data range**: The texture stores its data as real numbers between 0 and 1, while normal coordinates are in the range $-1$ to 1. Thus you need to apply a scale and then shift to the data before using it. The scale and shift should be chosen so that $0 \mapsto -1$ and $1 \mapsto 1$.

**Yet more matrix stuff**: We want to store our normal map data in such a way that we can use one data patch and tile it around a curvy surface (like a sphere or a mesh). As such, we don't want to represent the normal data in the texture as coordinates with respect to the world or object or joint or even bone frame.

For sake of notation, let $\vec{\mathbf{b}}^t = \vec{\mathbf{e}}^t M$ be the not necessarily orthonormal frame associated with a "bone" of your object (say the lower arm), about to be drawn, and $M$ the associated model view matrix.

We will let each vertex of this bone have its own "tangent frame" represented as $\vec{\mathbf{t}}^t = \vec{\mathbf{b}}^t T$. The data for $T$ will be passed as three `vec3` vertex attribute variables: `aTangent`, `aBinormal`, and `aNormal`. These will represent the three columns making up the upper left 3 by 3 submatrix of $T$. Since we will be dealing with the coordinates of vectors and not points, we will not need any translational data in $T$. In the texture, we will assume that the normal coordinates, $\mathbf{n} = [n_r, n_g, n_b, 0]^t$, are expressing the normal wrt to the $\vec{\mathbf{t}}^t$ frame. (When we want to apply the same texture to a surface oriented in some other direction, we just use a different $T$ matrix.)

In the vertex or fragment shader if we are given $\mathbf{v} := [x_e, y_e, z_e, 0]^t$ the eye coordinates of a vector that we want to dot with a normal, the dot should ultimately be calculated as

$$\mathbf{n}^t T^t M^{-1} \mathbf{v} = (M^{-t} T \mathbf{n})^t \mathbf{v}$$

When we do shading computation per fragment, more often than not we are interested in the dot between unit length vectors in the direction of the normal and some other vector, so we would calculate

$$\texttt{dot}(\texttt{normalize}(M^{-t} T \mathbf{n}), \texttt{normalize}(\mathbf{v}))$$

Since we will not get hold of the normal data until we get to the fragment shader, we will do the following: at the vertex shader we pass $M^{-t}T$ as a varying variable called `vNTMat` (for normal matrix times tangent frame matrix) to the fragment shader. Then at the fragment shader, you will multiply $\mathbf{n}$, the normal data fetched from the texture, by this matrix, and then normalize. The new vector is then used for various shading calculation.