MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.709 Computer Graphics

Lecture 04 - Lighting & Shading

Daniel Playne
d.p.playne@massey.ac.nz

Visual Appearance

When rendering images of a three-dimensional scene, it is important to not only have the correct geometrical shape but also the appropriate ***visual appearance***.

One of the most important steps in achieving ***visual realism*** is to model the interactions between light and the materials of the objects in the scene.

We will see how a simple (and commonly used) lighting and surface model can be implemented using OpenGL shaders to produce more convincing rendering.

Visual Phenomena

In a physical environment there are some phenomena that are useful to understand.

- Light is emitted by a natural or artificial light source.
- Light interacts with objects in the scene - absorbed, scattered or propagated in a new direction.
- Light is absorbed by a sensor (eye, film etc).

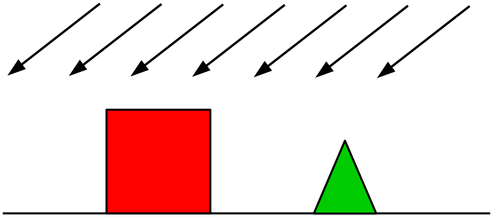
Light

Light itself is complicated and can be modelled in a number of different ways - geometric rays, electromagnetic waves or streams of photons. Regardless of this, a ***light source*** is something at ***emits*** light.

There are many different ways that light sources can be represented in computer graphics with varying degrees of complexity. In this lecture we will cover some simple approaches.

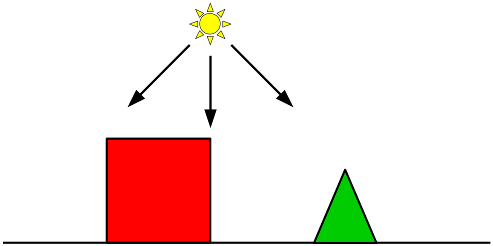
Light Sources

The simplest light source we will use is called a ***directional light*** which models an extremely distant light source (such as the sun) and considers light to be arriving at a constant direction across the entire scene.



Light Sources

Another light source that is commonly used a ***positional light*** which models a light source (usually as a point) within or close to the scene. For these light sources the light will be arriving from different directions depending on the position within the scene.



Light Sources

In addition to the direction the light, we also need to define the amount of illumination the light source emits.

The amount of light emitted by a light source is specified by the ***irradiance*** which is the power through a unit area surface perpendicular to the direction of the light.

This is the sum of the energy from all the photons passing through the surface in one second.

Light Sources

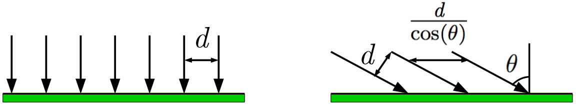
The light may also have a particular colour, in our simple lighting model this will consist of a simple RGB value (unlike colours, these RGB values for irradiance can be arbitrarily large).

While the RGB model is sufficient for specifying a colour, it does not correctly model how different frequencies of light may interact with different materials within a scene. There are some lighting models that try to simulate this more accurately.

Illumination

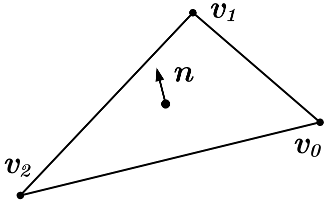
To calculate the illumination on a surface, we must compute the irradiance from all light sources at a plane parallel to that surface.

The irradiance on a surface E can be calculated as the irradiance measured perpendicular to the light direction E_L times the cosine of angle θ between $\hat{\mathbf{l}}$ (the light direction) and the normal of the surface $\hat{\mathbf{n}}$.



Surface Normals

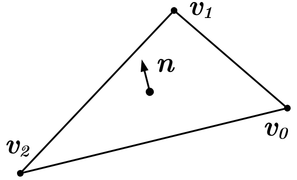
The ***surface normal*** (or ***face normal***) $\vec{\mathrm{n}}$ is a vector that is perpendicular to the surface. This is usually *normalised* to give a unit vector $\hat{\mathrm{n}}$.



Surface Normals

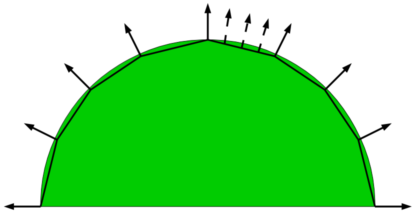
For a flat plane or primitive such as a triangle, the face normal can be calculated as follows:

$$\begin{aligned}\vec{\mathbf{e_1}} &= \vec{\mathbf{v_1}} - \vec{\mathbf{v_0}} \\ \vec{\mathbf{e_2}} &= \vec{\mathbf{v_2}} - \vec{\mathbf{v_0}} \\ \vec{\mathbf{n}} &= \vec{\mathbf{e_1}} \times \vec{\mathbf{e_2}} \\ \hat{\mathbf{n}} &= \frac{\vec{\mathbf{n}}}{|\vec{\mathbf{n}}|}\end{aligned}$$



Surface Normals

For a curved surface, the *surface normal* is perpendicular to the tangent plane at a particular point. The normals are defined at the vertex positions of the primitives approximating the curved surface and are then interpolated across the surface.



Illumination

The calculation of irradiance is clamped to non-negative values (the case where the light comes from behind the surface). This uses a cosine function that is clamped to a minimum value of 0 which is written as $\overline{\cos}(\theta)$.

$$\begin{aligned} E &= E_L \overline{\cos}(\theta) \\ &= E_L \mathsf{max}(\vec{\mathbf{n}} \cdot \vec{\mathbf{l}}, 0) \end{aligned}$$

Where the length of $\vec{\mathbf{n}}$ and $\vec{\mathbf{l}}$ are equal to 1 and we make use of the relationship $\vec{\mathbf{n}} \cdot \vec{\mathbf{l}} = |\vec{\mathbf{n}}||\vec{\mathbf{l}}| \cos(\theta)$.

Illumination

The irradiance on a surface from multiple light sources combine additively, this makes calculating the total irradiance on a surface as simple as adding up all of the irradiance values from each of the individual light sources:

$$E = \sum_{k=0}^{n} E_{L_k} \overline{\cos}(\theta_k)$$

Where E_{L_k} and θ_k are the irradiance and angle for the k^{th} light in the scene.

Materials

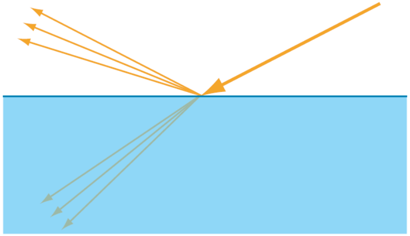
In most computer graphics rendering, objects are represented as surfaces. The appearance of these objects can be controlled by attaching a material to each of these surfaces that determine how the irradiance on the surface is ***absorbed*** and ***scattered***.

Scattering occurs when light encounters some discontinuity (a surface in our scenes), this does not change the amount of light but simply the direction.

Absorption happens when light is converted to some other form of energy inside matter. This does not change the direction but the amount of light.

Materials

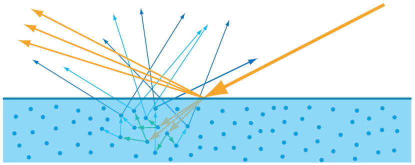
The most important optical discontinuity is the interface between the air and the surface of an object, at this interface the light is scattered in two directions - into the surface (***refraction***) and out of it (***reflection***).



RTR 3.05.06 www.realtimerendering.com / Copyright A.K. Peters Ltd. / Fair Use

Materials

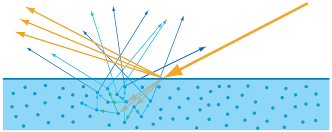
In a ***transparent*** object, the transmitted light continues to travel through the object. For now we will consider only objects that have multiple scattering and absorption events at (or very near) the surface and some light is re-emitted back away from the surface.



RTR 3.05.07 www.realtimerendering.com / Copyright A.K. Peters Ltd. / Fair Use

Materials

It can be seen from this illustration that the light reflected at the surface has a different direction than the light was the partially absorbed/scattered by the surface.



RTR 3.05.07 www.realtimerendering.com / Copyright A.K. Peters Ltd. / Fair Use

In our lighting calculations we will separate the surface lighting equations into two terms

- ***specular*** reflected light.
- ***diffuse*** transmitted-absorbed-scattered light.

Materials

Different materials may reflect/absorb different frequencies of light in different ratios. This is represented in our lighting model by using an RGB vector or colour for the ***specular*** and ***diffuse*** terms in our equations.

Adjusting the specular and diffuse colours for our materials allow us to represent materials of different compositions - steel, plastic, paper etc.

Phong Reflection Model

The ***Phong Reflection Model*** is a commonly used lighting model that produces a plausible approximation of lighting in a 3D scene.

Each object in a scene is assigned a material with three reflectance factors which have a different factor for red-green-blue components:

- ***ambient*** (k_a)
- ***diffuse*** (k_d)
- ***specular*** (k_s)

Phong Reflection Model

All light sources in a scene have the two intensities - ***diffuse*** and ***specular*** (also with different intensities for red-green-blue).

- ***diffuse*** (I_d)
- ***specular*** (I_s)

There is a single term I_a for the ***ambient*** light in a scene.

Phong Reflection Model

The ***ambient*** term represents light scattered around the entire scene, has no particular direction and lights all surfaces evenly.

Calculating the ambient light for a particular point on a surface is as simple as multiplying the ambient light term I_a by the ambient reflectance factor of the material k_a.

$$k_a \otimes I_a$$

(where $\otimes$ represents a piecewise multiplication of the rgb vectors)

Phong Reflection Model

The ***diffuse*** term represents the light from the light sources that is scattered off the surface of the object.

The diffuse light from a point on a surface can be calculated from the light's irradiance I_d, the direction towards the light $\vec{\mathbf{l}}$, the normal of the surface $\vec{\mathbf{n}}$ and the diffuse material k_d.

$$k_d(\vec{\mathbf{l}} \cdot \vec{\mathbf{n}}) \otimes I_d$$

(where $\vec{\mathbf{l}}$ and $\vec{\mathbf{n}}$ are unit vectors)

Phong Reflection Model

The ***specular*** term represents the light from the light sources that is reflected off the surface of the object.

The specular light from a point on a surface can be calculated from the light's irradiance I_d, the direction towards the light $\vec{\mathbf{l}}$, the normal of the surface $\vec{\mathbf{n}}$, the direction towards the viewer $\vec{\mathbf{v}}$, the specular material k_s and shininess coefficient α.

$$k_s(\vec{\mathbf{r}} \cdot \vec{\mathbf{v}})^\alpha \otimes I_s$$

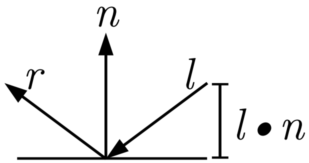
(where $\vec{\mathbf{l}}$, $\vec{\mathbf{n}}$ and $\vec{\mathbf{v}}$ are unit vectors and $\vec{\mathbf{r}}$ is the reflection of $\vec{\mathbf{l}}$ in the plane defined by $\vec{\mathbf{n}}$)

Calculating Reflected Vector

The reflected vector $\vec{\mathbf{r}}$ of a vector $\vec{\mathbf{l}}$ in a plane defined by $\vec{\mathbf{n}}$ can be calculated as follows (assuming $|\vec{\mathbf{l}}| = |\vec{\mathbf{n}}| = 1$):

$$\vec{\mathbf{r}} = \vec{\mathbf{l}} - 2(\vec{\mathbf{l}} \cdot \vec{\mathbf{n}})\vec{\mathbf{n}}$$

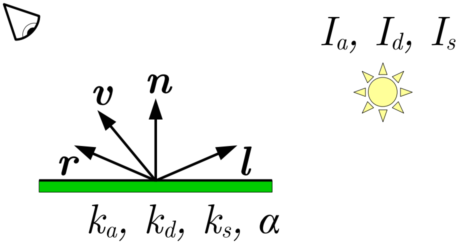
As illustrated below:



Phong Reflection Model

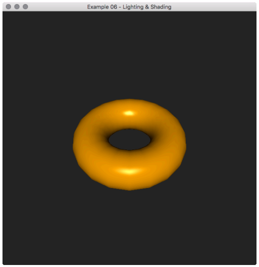
All put together, the Phong Reflection Model calculates the light intensity I_p at a point p on the surface of an object as

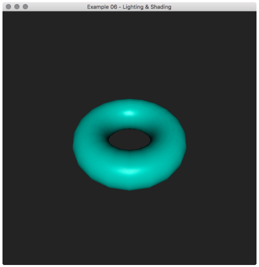
$$I_p = k_a \otimes I_a + k_d(\vec{\mathbf{l}} \cdot \vec{\mathbf{n}}) \otimes I_d + k_s(\vec{\mathbf{r}} \cdot \vec{\mathbf{v}})^{\alpha} \otimes I_s$$

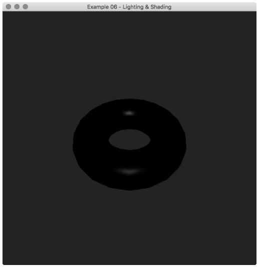


Phong Reflection Model

Some example lighting with three different materials - brass, turquoise and black plastic.







Shading

Shading is the process of using a lighting model to calculate the outgoing radiance I along a view direction $\vec{\mathbf{v}}$ based on the material properties and light sources.

We will look at two shading methods and how to implement them in GLSL shaders:

- Goraud Shading
- Phong Shading

Goraud Shading Model

The ***Goraud Shading Model*** is a method for calculating the colour for an object in a 3D scene based on a particular lighting model.

In Goraud Shading, the lighting is calculated per-vertex (in OpenGL it is performed in the vertex shader) and then the colour is interpolated across the primitive.

This was the default shading model used in early versions of DirectX and OpenGL as it is relatively cheap to compute (only one lighting calculation per vertex).

Goraud Shading Model - Vertex Shader

```glsl
// OpenGL 3.3
#version 330

// Input to Vertex Shader - Position and Normal
in vec4 vert_Position;
in vec4 vert_Normal;

// Transform Matrices
uniform mat4 u_Model;
uniform mat4 u_View;
uniform mat4 u_Projection;

// Light Source - Directional
uniform vec4 u_Light_Direction = vec4(0.0f, 0.0f, -1.0f, 0.0f);

// Light Source - Properties
uniform vec4 Ia = vec4(0.3f, 0.3f, 0.3f, 1.0f);
uniform vec4 Id = vec4(1.0f, 1.0f, 1.0f, 1.0f);
uniform vec4 Is = vec4(1.0f, 1.0f, 1.0f, 1.0f);

// Material - Brass
uniform vec4 Ka = vec4(0.329412, 0.223529, 0.027451, 1.0);
uniform vec4 Kd = vec4(0.780392, 0.568627, 0.113725, 1.0);
uniform vec4 Ks = vec4(0.992157, 0.941176, 0.807843, 1.0);
uniform float a = 27.89743616;

// Output to Fragment Shader
out vec4 frag_Colour;
```

Goraud Shading Model - Vertex Shader

```
void main() {
   // ────────── Calculate Vectors ──────────
   // Direction to Light (normalised)
   vec4 l = normalize(−(u_View ∗ u_Light_Direction));

   // Surface Normal (normalised)
   vec4 n = normalize(u_View ∗ u_Model ∗ vert_Normal);

   // Reflected Vector
   vec4 r = reflect(−l, n);

   // View Vector
   vec4 v = normalize(−(u_View ∗ u_Model ∗ vert_Position));

   // ────────── Calculate Terms ──────────
   vec4 Ta = Ka ∗ Ia;
   vec4 Td = Kd ∗ max(dot(l, n), 0.0) ∗ Id;
   vec4 Ts = Ks ∗ pow((max(dot(r, v), 0.0)), a) ∗ Is;

   // Vertex Colour
   frag_Colour = Ta + Td + Ts;

   // ────────── Vertex Position ──────────
   gl_Position = u_Projection ∗ u_View ∗ u_Model ∗ vert_Position;
}
```

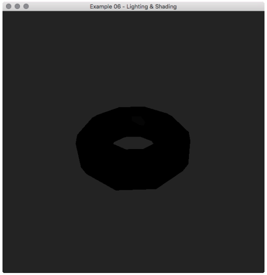
Goraud Shading Model

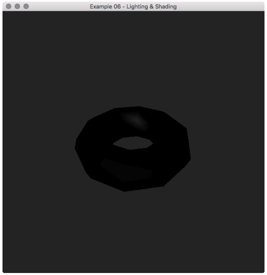
While the Goraud Shading model does a reasonably good job of computing lighting, it does suffer from some problems:

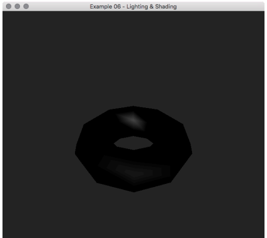
- Loss of detail in surface centres (specular highlights cannot be centred between vertices).
- Colours are linearly interpolated rather than normals.
- Specular highlights are highly dependent on view and light directions and flash with motion.

Goraud Shading Model

For example, the following three images are of the same object with slightly different rotations.







Phong Shading Model

The ***Phong Shading Model*** is not to be confused with the ***Phong Reflection Model*** (although they were published together and are both named after their author Bui Tuong Phong).

The ***Phong Shading Model*** is a method for calculating shading the surface of an object. In this model, the lighting is calculated *per-fragment* rather than *per-vertex* as in the Goraud shading model.

Phong Shading Model

The Phong Shading model interpolates the surface normals between vertexes and then computes the lighting model for each individual fragment using the interpolated normal.

This has the advantage of producing more realistic surface shading (especially for smoothly curved surfaces) and avoids improper interpolation of colours.

However, it does come at the cost of being significantly more expensive to compute.

Phong Shading Model - Vertex Shader

```glsl
// OpenGL 3.3
#version 330

// Input to Vertex Shader
in vec4 vert_Position;
in vec4 vert_Normal;

// Transform Matrices
uniform mat4 u_Model;
uniform mat4 u_View;
uniform mat4 u_Projection;

// Light Source - Directional
uniform vec4 u_Light_Direction = vec4(0.0f, 0.0f, -1.0f, 0.0f);

// Output to Fragment Shader
out vec4 frag_Position;
out vec4 frag_Normal;
out vec4 frag_Light_Direction;

void main() {
   // Output to Fragment Shader - Position, Normal and Light Direction
   frag_Position        = u_View * u_Model * vert_Position;
   frag_Normal          = u_View * u_Model * vert_Normal;
   frag_Light_Direction = u_View           * u_Light_Direction;

   // ---------- Vertex Position ----------
   gl_Position = u_Projection * u_View * u_Model * vert_Position;
}
```

Phong Shading Model - Fragment Shader

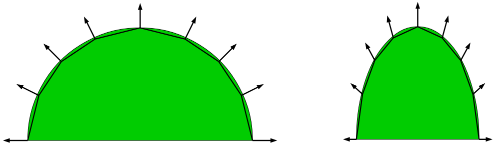
```
// OpenGL 3.3
#version 330

// Input from Vertex Shader
in vec4 frag_Position;
in vec4 frag_Normal;
in vec4 frag_Light_Direction;

// Output from Fragment Shader
out vec4 pixel_Colour;

// Light Source
uniform vec4 Ia = vec4(0.3f, 0.3f, 0.3f, 1.0f);
uniform vec4 Id = vec4(1.0f, 1.0f, 1.0f, 1.0f);
uniform vec4 Is = vec4(1.0f, 1.0f, 1.0f, 1.0f);

// Material – Brass
uniform vec4 Ka = vec4(0.329412, 0.223529, 0.027451, 1.0);
uniform vec4 Kd = vec4(0.780392, 0.568627, 0.113725, 1.0);
uniform vec4 Ks = vec4(0.992157, 0.941176, 0.807843, 1.0);
uniform float a = 27.89743616;
```

Phong Shading Model - Fragment Shader

```
void main () {
   // ———————— Calculate Vectors ————————
   // Direction to Light (normalised)
   vec4 l = normalize(−frag_Light_Direction);

   // Surface Normal (normalised)
   vec4 n = normalize(frag_Normal);

   // Reflected Vector
   vec4 r = reflect(−l, n);

   // View Vector
   vec4 v = normalize(−frag_Position);

   // ———————— Calculate Terms ————————
   vec4 Ta = Ka ∗ Ia;
   vec4 Td = Kd ∗ max(dot(l, n), 0.0) ∗ Id;
   vec4 Ts = Ks ∗ pow((max(dot(r, v), 0.0)), a) ∗ Is;

   // ———————— Fragment Colour ————————
   pixel_Colour = Ta + Td + Ts;
}
```

Transforming Normal Vectors

It is important to take note of some aspects of transforming normal vectors. The normals are direction vectors so translations should not affect them and if the scaling of the model and view matrix are all uniform then the normals can simply be transformed by the ***Model*** and ***View*** transforms.



However, if there is non-uniform scaling then the normal vectors could be distorted by the transform.

Transforming Normal Vectors

If there is non-uniform scaling in the ***Model*** or ***View*** transforms then a separate ***normal*** transform (N) should be computed to transform the normals separately. This can be calculated from:

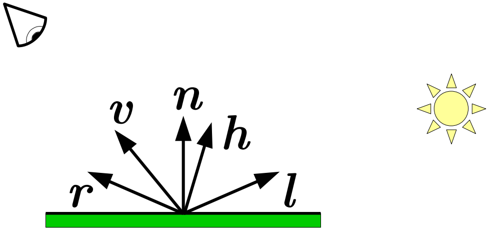
$$N = (M^{-1})^T$$

The inverse of a pure rotation transform is the same as its transpose. The inverse of a scale transform is the opposite while the transpose is the same. The inverse-transpose of a combined scale and rotation matrix inverts the scale but leaves the rotation unchanged.

Blinn-Phong Reflection Model

The ***Blinn-Phong Reflection Model*** is a modified version of the Phong Reflection Model that is optimised to avoid recalculation of the reflection vector for each vertex or fragment.

Instead the Blinn-Phong Reflection model uses another *halfway* vector that is between the viewer and light-source vectors.



Blinn-Phong Reflection Model

The halfway vector $\vec{\mathbf{h}}$ can be calculated from:

$$\vec{\mathbf{h}} = \frac{\vec{\mathbf{l}} + \vec{\mathbf{v}}}{|\vec{\mathbf{l}} + \vec{\mathbf{v}}|}$$

The term $\vec{\mathbf{r}} \cdot \vec{\mathbf{v}}$ in the Phong reflection model then replaced with $\vec{\mathbf{n}} \cdot \vec{\mathbf{h}}$ to give:

$$I_p = k_a \otimes I_a + k_d(\vec{\mathbf{l}} \cdot \vec{\mathbf{n}}) \otimes I_d + k_s(\vec{\mathbf{n}} \cdot \vec{\mathbf{h}})^\alpha \otimes I_s$$

Blinn-Phong Reflection Model

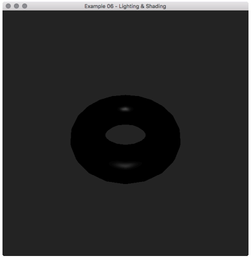
The angle between the halfway vector and the surface normal is likely to be smaller than the angle between the reflected vector and the viewer (unless viewed from a very steep angle).

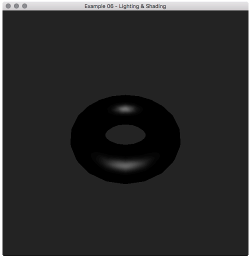
To correct for this, a different exponent α' can be used such that $(\vec{\mathbf{n}} \cdot \vec{\mathbf{h}})^{\alpha'}$ is closer to $(\vec{\mathbf{r}} \cdot \vec{\mathbf{v}})^{\alpha}$.

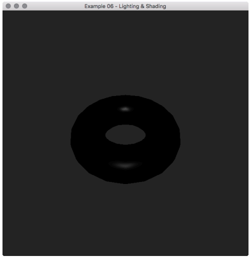
For front-light objects a value of $\alpha' = 4\alpha$ gives similar results to the Phong Reflection model.

Blinn-Phong Reflection Model

Examples of the Phong reflection model (left), the Blinn-Phong reflection model with $\alpha' = \alpha$ (centre) and the Blinn-Phong reflection model with $\alpha' = 4\alpha$ (right).







GLM - OpenGL Mathematics

OpenGL Mathematics or GLM is a header-only C++ library based on the GLSL shading language specifications. We will use this library to store and manipulate vectors and matrices on the CPU as it provides implementations for many common operations and makes manipulating these types easy.

GLM - OpenGL Mathematics

Where possible GLM is consistent with the types in GLSL shaders:

Type	Description
`vec2`	2-component, floating-point vector
`vec3`	3-component, floating-point vector
`vec4`	4-component, floating-point vector
`ivec2`	2-component, integer vector
`ivec3`	3-component, integer vector
`ivec4`	4-component, integer vector
`mat2`	2x2 floating-point matrix
`mat3`	3x3 floating-point matrix
`mat4`	4x4 floating-point matrix

GLM - OpenGL Mathematics

GLM provides extra functionality above and beyond GLSL features with a range of extensions as organised below:

- ***GLM Core*** - Only the GLSL specification.
- ***GTC Extensions*** - Functions and Types not specified by GLSL (stable).
- ***GTX Extensions*** - Experimental Functions and Types not specified by GLSL (unstable).

For this course we will stick to using ***GLM Core*** and the stable ***GTC Extensions***.

GLM - OpenGL Mathematics

Example: building a model matrix with separate rotations around the X-axis and Y-axis.

```cpp
// GLM Headers
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

...

// ——————————————————————————————————————————
// Model Matrix
glm::mat4 modelMatrix = glm::mat4(1.0f);
float modelThetaX = 0.8f;
float modelThetaY = 0.0f;

// Calculate Model Matrix
modelMatrix = glm::rotate(modelMatrix, modelThetaX, glm::vec3(1.0f, 0.0f, 0.0f));
modelMatrix = glm::rotate(modelMatrix, modelThetaY, glm::vec3(0.0f, 1.0f, 0.0f));

// Get Model Matrix location
GLint modelLoc = glGetUniformLocation(program, "u_Model");

// Copy Rotation Matrix to Shader
glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(modelMatrix));
// ——————————————————————————————————————————
```

GLM - OpenGL Mathematics

Example: building a view matrix using the GLM function `lookAt`.

```cpp
// ———————————————————————————————————————
// View Matrix
glm::mat4 viewMatrix;
glm::vec3 viewPosition(0.0f,  0.0f,  5.0f);
glm::vec3 viewUp      (0.0f,  1.0f,  0.0f);
glm::vec3 viewForward (0.0f,  0.0f, -1.0f);

// Normalise Vectors
viewUp      = glm::normalize(viewUp);
viewForward = glm::normalize(viewForward);

// Construct View Matrix
viewMatrix = glm::lookAt(viewPosition, viewPosition + viewForward, viewUp);

// Get View Matrix location
GLint viewLoc = glGetUniformLocation(program, "u_View");

// Copy View Matrix to Shader
glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(viewMatrix));
// ———————————————————————————————————————
```

GLM - OpenGL Mathematics

Example: building a perspective projection matrix.

```cpp
// ----------------------------------------
// Projection Matrix
glm::mat4 projectionMatrix;

// Calculate Perspective Projection
projectionMatrix = glm::perspective(glm::radians(67.0f), 1.0f, 0.2f, 10.0f);

// Get Projection Matrix location
GLint projectionLoc = glGetUniformLocation(program, "u_Projection");

// Copy Projection Matrix to Shader
glUniformMatrix4fv(projectionLoc, 1, GL_FALSE, glm::value_ptr(projectionMatrix));
// ----------------------------------------
```

GLM - OpenGL Mathematics

Example: creating a cube (top face)

```cpp
// ---------------------------------------------------------------------------------
// Create Cube with Positions and Normals
void createCube(std::vector<glm::vec4> &buffer,
                std::vector<glm::ivec3> &indexes) {
   // ---- Top ----
   // Left-Top-Back - 0
   buffer.push_back(glm::vec4(-1.0f,  1.0f, -1.0f,  1.0f));
   buffer.push_back(glm::vec4( 0.0f,  1.0f,  0.0f,  0.0f));

   // Right-Top-Back - 1
   buffer.push_back(glm::vec4( 1.0f,  1.0f, -1.0f,  1.0f));
   buffer.push_back(glm::vec4( 0.0f,  1.0f,  0.0f,  0.0f));

   // Left-Top-Front - 2
   buffer.push_back(glm::vec4(-1.0f,  1.0f,  1.0f,  1.0f));
   buffer.push_back(glm::vec4( 0.0f,  1.0f,  0.0f,  0.0f));

   // Right-Top-Front - 3
   buffer.push_back(glm::vec4( 1.0f,  1.0f,  1.0f,  1.0f));
   buffer.push_back(glm::vec4( 0.0f,  1.0f,  0.0f,  0.0f));

   // Top
   indexes.push_back(glm::ivec3(0, 2, 3));
   indexes.push_back(glm::ivec3(0, 3, 1));
   ...
```