COMP3421
The programmable pipeline and Shaders
The graphics pipeline

Model

Model-View Transform

Model Transform → View Transform

Projection transformation

Illumination

Perspective division

Clipping

Viewport

Hidden surface removal

Rasterisation

Frame buffer

Display

Texturing

User
The graphics pipeline

Model

Model-View Transform

Model Transform → View Transform → Vertex shading → Projection transformation

Rasterisation

Viewport

Perspective division

Clipping

Fragment shading

Hidden surface removal

Frame buffer

Display

User
Shading

Illumination (a.k.a shading) is done at two points in the fixed function pipeline:

- **Vertices** in the model are shaded before projection.
- **Pixels** (fragments) are shaded in the image after rasterisation.
- Doing more work at the **vertex** level is more efficient. Doing more work at the **pixel** level can give better results.
Vertex shading

The built-in lighting in OpenGL is mostly done as vertex shading.

The lighting equations are calculated for each vertex in the image using the associated vertex normal.

Illumination is calculated at each of these vertices.
Vertex shading

The normal vector \( \mathbf{m} \) used to compute the lighting equation is the normal we specified when creating the vertex using

\[
gl.glNormal3d(mx, my, mz);
\]
Vertex shading

This is why we use different normals on curved vs flat surfaces, so the vertex may be lit properly.
Vertex shading

Illumination values are attached to each vertex and carried down the pipeline until we reach the fragment shading stage.

```c
struct vert {
    float[4] pos;  // vertex coords
    float[4] light; // rgba colour
    // other info...
}
```
Knowing the vertex properties, we need calculate appropriate colours for every pixel that makes up the polygon.

There are three common options:

- **Flat** shading
- **Gouraud** shading
- **Phong** shading
In OpenGL

// Flat shading :

gl.glShadeModel(GL2.GL_FLAT);

// Gouraud shading (default):

gl.glShadeModel(GL2.GL_SMOOTH);

// Phong shading:

// No built-in implementation
Flat shading

The simplest option is to shade the entire face the same colour:

• Choose one vertex (arbitrarily chosen by OpenGL…)
• Take the illumination of that vertex
• Set every pixel to that value.
Flat shading

Flat shading is good for:

• Diffuse illumination
• for flat surfaces
• with distant light sources

It is the fastest shading option.
Flat shading

Flat shading is bad for:

- close light sources
- specular shading
- curved surfaces
- Edges between faces become more pronounced than they actually are (Mach banding)

![Diagram showing close source and curved surface]
Gouraud shading is a simple smooth shading model.

We calculate fragment colours by bilinear interpolation on neighbouring vertices.

\[
\begin{align*}
\text{colour}(R_1) &= lerp(V_1, V_2, \frac{y-y_1}{y_2-y_1}) \\
\text{colour}(R_2) &= lerp(V_3, V_4, \frac{y-y_3}{y_4-y_3}) \\
\text{colour}(P) &= lerp(R_1, R_2, \frac{x-x_1}{x_2-x_1})
\end{align*}
\]
Gouraud shading is good for:

- curved surfaces
- close light sources
- diffuse shading

![Diagram](curved_surface_varying_normal_varying_diffuse_illumination.png)
Gouraud shading

Gouraud shading is only slightly more expensive than flat shading.

It handles specular highlights **poorly**.

- It works if the highlight occurs at a vertex.
- If the highlight would appear in the middle of a polygon it disappears.
- Highlights can appear to jump around from vertex to vertex with light/camera/object movement.
Phong shading

Phong shading is designed to handle specular lighting better than Gouraud. It also handles diffuse better as well.

It works by deferring the illumination calculation until the fragment shading step.

So illumination values are calculated per fragment rather than per vertex.

Not implemented on the fixed function pipeline. Need to use the programmable pipeline.
Phong shading

For each fragment we need to know:

- source vector $s$
- eye vector $v$
- normal vector $m$

Knowing the source location, camera location and fragment location we can compute $s$ and $v$.

What about $m$?
Normal interpolation

Phong shading approximates $m$ by interpolating the normals of the polygon.

Vertex normals
Normal interpolation

Phong shading approximates $\mathbf{m}$ by interpolating the normals of the polygon.
Normal interpolation

In a 2D polygon we do this using (once again) **bilinear interpolation**.

However the interpolated normals will vary in length, so they need to be **normalised** (set length = 1) before being used in the lighting equation.
Phong shading

Pros:

• Handles specular lighting well.
• Improves diffuse shading
• More physically accurate
Phong shading

Cons:

• Slower than Gouraud as normals and illumination values have to be calculated per pixel rather than per vertex. In the old days this was a BIG issue. Now it is usually ok.
Fixed function pipeline

Vertex transformations

Model

Model-View Transform

Model Transform → View Transform

Illumination

Projection transformation

Clipping

Perspective division

Viewport

Hidden surface removal

Frame buffer

Display

User

Rasterisation

Texturing

Fragment transformations

Model-View Transform

Model Transform → View Transform

Illumination
Programmable pipeline

Vertex transformations

Vertex Shader

Rasterisation

Viewport

Perspective division

Clipping

Hidden surface removal

Frame buffer

Display

User

Fragment shader

Fragment transformations
Shaders

The programmable pipeline allows us to write shaders to perform arbitrary vertex and fragment transformations. (There are also optional tessellation and geometry shaders.)

Shaders are written in a special language called GLSL (GL Shader Language) suitable for specifying functions on the GPU.
Basic Vertex Shader

/* This does the bare minimum. It transforms the input gl_Vertex and outputs the gl_Position value in 4DCVVV coordinates */

void main(void) {

    gl_Position =

    gl_ModelViewProjectionMatrix *

    gl_Vertex;

}
Basic Fragment Shader

/* Make every fragment red */

void main (void) {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}

GPU

The graphics pipeline performs the same transformations to thousands of millions of vertices and fragments.

These transformations are highly parallelisable.

The GPU is a large array of SIMD (single instruction, multiple data) processors.
Vertex Shaders

Replaces fixed function

- vertex transformation
- normal transformation, normalization
- vertex illumination
- Has access to OpenGL states such as model view transforms, colors etc (in later versions these must be explicitly passed in). These variables start with gl_ (eg gl_Color)
Vertex Shaders

**Input:** individual vertex in model coordinates

**Output:** individual vertex in clip (4Dcvv) coordinates

They may also be sent other inputs from the application program and output color, lighting and other values for the fragment shader.

They operate on individual vertices and results can't be shared with other vertices.

They can't create or destroy a vertex.
Fragment Shaders

Replaces fixed function

- texturing and colouring the fragment.

Enables lighting to be done at the fragment stage (such as phong shading) which could not be done in fixed function pipeline

Has access to OpenGL states (in later versions this must be explicitly passed in)
Fragment Shaders

**Input:** individual fragment in window coordinates

**Output:** individual fragment in window coordinates (with color set)

May receive inputs (that may be interpolated) from the vertex shader and inputs from the application program.

They can’t share results with other fragments.

Can access and apply textures.
Fragment Shaders

The fragment shader does not replace the fixed functionality graphics operations that occur at the back end of the OpenGL pixel processing pipeline such as

• depth testing

• alpha blending.
Setting up Shaders

1. Create one or more empty shader objects with `glCreateShader`.
2. Load source code, in text, into the shader with `glShaderSource`.
3. Compile the shader with `glCompileShader`.
4. Create an empty program object with `glCreateProgram`. This returns an int ID.
5. Bind your shaders to the program with `glAttachShader`.
6. Link the program with `glLinkProgram`.
7. See Shader.java for details
Setting up Shaders in OpenGL code

// create shader objects and
// reserve shader IDs

int vertShader =
gl.glCreateShader(GL2.GL_VERTEX_SHADER);

int fragShader =
gl.glCreateShader(GL2.GL_FRAGMENT_SHADER);
OpenGL

// compile shader which is just
// a string (do once for vertex
// and then for fragment)

gl.glShaderSource(
    vertShader,
    vShaderSrc.length,
    vShaderSrc, vLengths, 0);
gl.glCompileShader(vertShader);
OpenGL

// check compilation

int[] compiled = new int[1];

gl.glGetShaderiv(vertShader,
    GL2ES2.GL_COMPILE_STATUS,
    compiled, 0);

if (compiled[0] == 0) {
    // compilation error!
}
OpenGL

// program = vertex shader + frag shader

int shaderprogram =
    gl.glCreateProgram();

    gl.glAttachShader(shaderprogram,
        vertShader);

    gl.glAttachShader(shaderprogram,
        fragShader);

    gl.glLinkProgram(shaderprogram);

    gl.glValidateProgram(shaderprogram);
Using Shaders

After your shaders have been set up you need to tell OpenGL what shaders to use.

OpenGL uses the current shader program.

To set the current shader use:

```c
gl.glUseProgram(shaderProgramID);
```
Using Shaders

Can set multiple shaders within a program.

```cpp
gl.glUseProgram(shaderProgram1);
//render an object with shaderprogram1

gl.glUseProgram(shaderProgram2);
//render an object with shaderprogram2

gl.glUseProgram(0);
//render an object with fixed function pipeline
```
Color Data

**JOGL**

```java
gl.glColor3f
```

**VertexShader**

```java
glFrontColor = gl_Color;
glBackColor = gl_Color;
```

**FragmentShader**

```java
glFragColor = gl_Color;
glFragColor = gl_Color;
```

Interpolated
void main(void) {
    gl_Position =
    gl_ModelViewProjectionMatrix *
    gl_Vertex;
    gl_FrontColor = gl_Color;
    gl_BackColor = gl_Color;
}
Passthrough Fragment Shader

/* fragment shader */

void main(void) {

/* gl_FrontColor and gl_BackColor are set by each vertex in the vertex shader and the rasteriser interpolates the appropriate value to get the fragment gl_Color */

    gl_FragColor = gl_Color;

}
Interpolation

By default the rasteriser interpolates vertex attributes such as

positions, colors, normals

across primitives to generate the right interpolated values for fragments.

We can turn on flat shading in jogl or use the keyword `flat` in our own shader variables to stop interpolation.
GLSL Syntax

C like language with

• No long term memory
• Basic types: float int bool
• Other type: sampler (textures)
• C++ Style Constructors
• Standard C/C++ arithmetic and logic operators
• if statements, loops
GLSL Syntax

Has limited support for loops

```cpp
for(i=0; i< n; i++){
    /* etc */
}
```

Conditional branching is much more expensive than on CPU!

Do not use too much – especially in fragment shader (there are usually lots of fragments!)
GLSL Syntax

Has support for 2D, 3D, 4D vectors (array like list like containers) of different types

- `vec2`, `vec3`, `vec4` are float vectors.
- `ivec2`, `ivec3`, `ivec4` are int vectors.

Has support for float matrix types

- `mat2`, `mat3`, `mat4`

Operators are overloaded for matrix and vector operations
GLSL Syntax

No characters, strings or printf

No pointers

No recursion

No double (limited support in later versions)

May not cast implicitly eg

vec3(1,0,1) may NOT work as it needs float. vec3(1.0,0.0,1.0) is correct.
Vectors

C++ Style Constructors

```cpp
vec3 a = vec3(1.0, 2.0, 3.0);
```

For vectors can use [], xyzw, rgba

```cpp
vec3 v;
```

`v[1]`, `v.y`, `v.g` all refer to the same element

Swizzling:

```cpp
vec3 a, b;
```

```cpp
a.xy = b.yx;
```
Matrix Components

Matrices are in column major order

M[i][j] is column i row j

mat4 m = mat4(1.0); //identity matrix
m = mat4(1.0,2.0,3.0,4.0, //first col
         5.0,6.0,7.0,8.0, //second col
         9.0,10.0,11.0,12.0, //third col
        13.0,14.0,15.0,16.0); //fourth col

float f = m[0][1]; //Would be 2.0
Variable Qualifiers

**uniform**: read-only input variables to vertex or fragment shader. Can’t be changed for a given primitive.

**Attributes**: May be different for each vertex.

**in**: read-only input variables to vertex shader.

**out, in**: Outputs from the vertex shader that are passed in to the fragment shader. They are interpolated for each fragment.
GLSL Compatability

Macs may not have version #130
Use #120 and change
in to attribute in your vertex shader
out to varying in your vertex shader
in to varying in your fragment shader
Built in Vertex Shader

Attributes

in: gl_Vertex,
   gl_Normal,
   gl_Color
out: gl_FrontColor,
    gl_BackColor,
    gl_Position (must be written)
Built in Fragment Shader Variables

in: gl_FragCoord

  gl_Color - If this is a front facing it will be the interpolated value set by the vertex shader for gl_FrontColor, (if it is back facing it will be gl_BackColor Note: gl_BackColor does not work on my computer).

out: gl_Fragment (fragment colour)

in/out: gl_depth
Built-in Uniform Variables

`gl_ModelViewMatrix`
`gl_ModelViewProjectionMatrix`,
`gl_NormalMatrix`
`gl_LightSource[0].position` (in camera coords)
`gl_LightSource[0].diffuse`
`etc`
Aside: gl_NormalMatrix

If the *modelview* matrix contains a non-uniform scale then it will not transform normals properly. It is no longer perpendicular!
Instead we use the transpose of the inverse of the upper left 3*3 corner of the modelview matrix.

The fixed function pipeline has been doing this behind the scenes for us.

We can use gl_NormalMatrix

Derivation (not examinable)
void main() {
    // Ambient light calculations
    vec4 globalAmbient =
        gl_LightModel.ambient * 
        gl_FrontMaterial.ambient;
    vec4 ambient =
        gl_LightSource[0].ambient * 
        gl_FrontMaterial.ambient;
Garoud vertex shader

// Diffuse calculations
vec3 v, normal, lightDir;

// transform the normal into eye space and normalize
normal = normalize(gl_NormalMatrix * gl_Normal);

// transform co-ords into eye space
v = vec3(gl_ModelViewMatrix * gl_Vertex);

// Assuming point light, get a vector TO the light
lightDir = normalize(gl_LightSource[0].position.xyz - v);

float NdotL = max(dot(normal, lightDir), 0.0);

vec4 diffuse = NdotL * gl_FrontMaterial.diffuse * 
gl_LightSource[0].diffuse;
vec4 specular = vec4(0.0,0.0,0.0,1.0);
vec3 dirToView = normalize(-v);
vec3 H = normalize(dirToView+lightDir);

// compute specular term if NdotL is larger than zero
if (NdotL > 0.0) {
    float NdotHV = max(dot(normal, H),0.0);
    specular = gl_FrontMaterial.specular *
               gl_LightSource[0].specular *
               pow(NdotHV,gl_FrontMaterial.shininess);
}
Garoud vertex shader

```cpp
gl_FrontColor = gl_FrontMaterial.emission +
    globalAmbient +
    ambient +
    diffuse +
    specular;

gl_Position = gl_ModelViewProjectionMatrix *
    gl_Vertex;
```
Garoud Light
fragment shader

//gl_Color is calculated in the vertex shader and interpolated

void main() {
    gl_FragColor = gl_Color;
}

Phong vertex shader

/* data associated with each vertex */

out vec3 N; //Send eyeCoords normal to fragment shader
out vec3 v; //Send eyeCoords position to fragment shader

void main(){

    /* send point and the normal in camera coords to the fragment shader – these will be interpolated */

    v = vec3(gl_ModelViewMatrix * gl_Vertex);
    N = normalize(gl_NormalMatrix * gl_Normal);

    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}


Phong fragment shader

in vec3 N;  //Interpolated from vertex shader outputs
in vec3 v;  //Interpolated from vertex shader outputs

void main (void){
    vec3 lightDir =
        normalize(gl_LightSource[0].position.xyz - v);
    vec3 dirToView = normalize(-v);
    vec3 normal = normalize(N);

    //etc same calculations as done in the
    // specular vertex shader
    // except we are doing calculations on EVERY fragment
    gl_FragColor = globalAmbient + ambient + diffuse +
    specular;
}

Things to try

These shaders have not handled basics such as

- multiple lights
- two sided lighting
- lights being enabled/disabled
- directional lights, spotlights etc
- attenuation...
More...

There are many more shading algorithms designed to implement different lighting techniques with different levels of speed and accuracy.

For example Cook Torrance is a more realistic model than Phong or Blinn-Phong.

Check out the Graphics Gems and GPU Gems books for lots of ideas.
User Defined Variables

To pass in your own uniforms or attribute ‘in’ variables into your shaders from the application program

```c
int loc =
gl.glGetUniformLocation(shaderProgram,"myVal");
gl.glUniform1f(loc, 0.5);
//Or for attributes
int loc =
glGetAttribLocation(shaderProgram,"myVal");
gl.glVertexAttrib1f(loc, 1, 1, 0, 1);
```
Optional Shaders

In later versions on GLSL, there are optional shaders between the vertex shader and the clipping stage.

**Tesselation Shaders**: can create additional vertices in your geometry

**Geometry Shader**: can be used to add, modify, or delete geometry,
GLSL Documentation

For the version compatible with class demos and slides: