Game Graphics & Real-time Rendering

CMPM 163, S2019

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Manu Thomas (TA)
mthomas6@ucsc.edu

David Abramov (TA)
dabramov@ucsc.edu

Website: creativecoding.soe.ucsc.edu/courses/cmpm163_s19
Slack: https://ucsccmpm163.slack.com

Class information

Class website:

https://creativecoding.soe.ucsc.edu/courses/cmpm163_s19

Slack is the main form of class communication:

https://ucsccmpm163.slack.com

Our TAs are Manu Thomas and David Abramov

They will lead the lab sections starting next week

uniform: data that is static over the life of the binding – It will be the same for all geometry passed in the shader. Unity defines many built-in uniforms via the **"UnityCG.cginc"** include within the shader. Uniform variables are available to both the vertex and the fragment shader.

See https://docs.unity3d.com/Manual/SL-UnityShaderVariables.html for the list of built-in uniform variables.

You can define your own uniforms within the Properties block and the top of the Unity shader program:

```
Properties
{
_MyCustomUniformVar (“MyFloatVal", Float) = 10
}
```

And then making sure to define a uniform in the Pass (where the vertex and fragment shader code is placed

```
uniform float _ MyCustomUniformVar;
```

(see any of the demos for Week2 for examples)

```
Properties
{
_MyCustomUniformVar ("MyFloatVal ", Float) = 10
}
```

This code adds an editable field that expects a float value (here, titled MyFloatVal, but it could be named anything you want) to the inspector when you select the shader from within Unity.

```
uniform float _ MyCustomUniformVar;
```

This line sets up a datalink from Unity to the shader program and allows you to access that value from within your vertex shader or fragment shader

attribute: data that is defined *per vertex* for each geometry object that you pass in (i.e., position, normal, texture coordinates). Unity defines most of this for you automatically.

You specify what attribute data you need using a special structure called “appdata”. This is how it commonly looks:

```
struct appdata
{
float4 vertex : POSITION;
float3 normal: NORMAL;
float2 uv : TEXCOORD0;
};
```

These attribute variables are then directly available in the vertex shader.

varying: this is how you link data from the vertex shader to the fragment shader. Varying data passed into the fragment shader is automatically *interpolated* across the triangle. E.g., the values for a pixel at the very center of a triangle would be an average of the values at the vertices.

You define the varying data with a struct, and make sure that your vertex shader outputs this struct.

```
struct v2f {
    float4 vertex : SV_POSITION;
    float3 normal : NORMAL;
};
v2f vert (appdata v) {
    v2f o;
    // do stuff...
    return o;
}
fixed4 frag (v2f i) : SV_Target {
    //do stuff...
    return float4(1,0,0,1); //or whatever you want the color to be
}
```

Phong Lighting in a Unity shader

A commonly used shader is called a "Phong shader," named after Bui Tuong Phong, a computer scientist who studied at University of Utah (where much of computer graphics was invented). It approximates ambient, diffuse, and specular lighting.

As before, it takes each triangle used to define your geometry and projects it into 2D, then passes this 2D data to the fragment shader.

But each triangle is colored depending on the interaction between: the camera orientation, the surface normal, and the light position

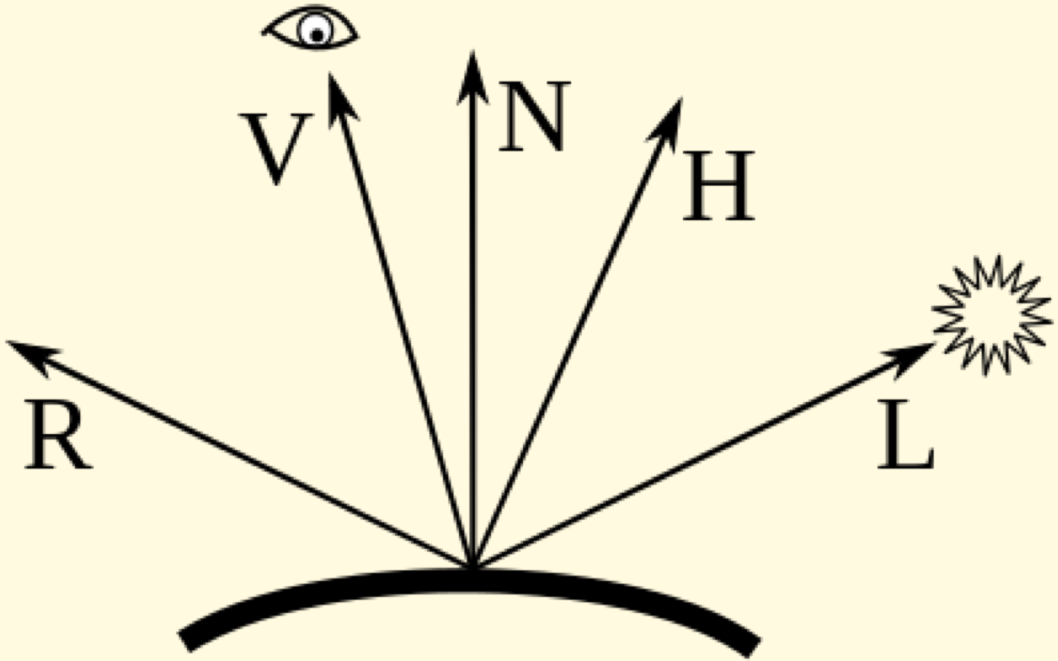
Phong shading

As before, this shader it takes each triangle used to define your geometry and projects it into 2D, then passes this 2D data to the fragment shader.

But each triangle is then colored *depending on the interaction between*: the camera orientation, the surface normal, and the light position.

- The camera position and orientation is defined by your renderer
- The surface normal is created automatically by Blender, or any modeling software will also create these for you
- The point light position is placed by you in Unity and automatically passed into the shader program

Blinn-Phong lighting



Phong shading

In the diagram above, the curved arc represents some mesh in your scene, and the vectors are emanating from a particular pixel within a particular triangle of that mesh.

The V (for **view**) vector points to the camera

The L (for **light**) vector points to a point light

The N (for **normal**) is the surface normal at the pixel

The H (for **half**) is a vector halfway between the V an L vectors, and is used to calculate specular highlights

Phong shading

In Demo code (adapted from The CG Tutorial, linked to on the class website), lighting is calculated "per-pixel" (in the fragment shader), in *world* coordinates.

Remember, the vertex shader is responsible for taking vertex in 3D coordinates into 2D coordinates. A coordinate system defines the objects in the scene according to a particular origin, and you can "transform" from one coordinate system to another using matrix multiplication.

Phong shading

The vertex shader is REQUIRED to transform the vertices into “clip” space, which can be done with the following Unity function:

o.vertex = UnityObjectToClipPos(xyz);

(see https://docs.unity3d.com/Manual/SL-BuiltinFunctions.html for Unity’s built-in shader functions)

But we also can perform other transformations and pass them into the fragment shader:
o.vertexInWorldCoords = mul(unity_ObjectToWorld, v.vertex);

(unity_ObjectToWorld is a built-in matrix provided by Unity)

This transforms the vertex into “world” coordinates, and places it into the structure containing the **varying** data, which is then available to the fragment shader

Phong shading

Now we can set up all of the vectors needed for lighting calculations in world space within the fragment shader:

```
fixed4 frag(v2f i) : SV_Target {
    float3 P = i.vertexInWorldCoords.xyz;
    float3 N = normalize(i.normal);
    float3 V = normalize(_WorldSpaceCameraPos - P);
    float3 L = normalize(_WorldSpaceLightPos0.xyz - P);
    float3 H = normalize(L + V);
    //do diffuse and specular lighting calculations …
}
```

(_WorldSpaceCameraPos and _WorldSpaceLightPos0 are provided automatically by Unity). See the phongDemo in the week2 code demos (available on the class website).