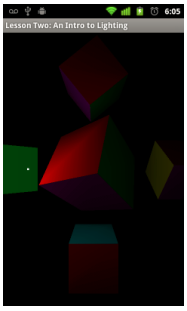
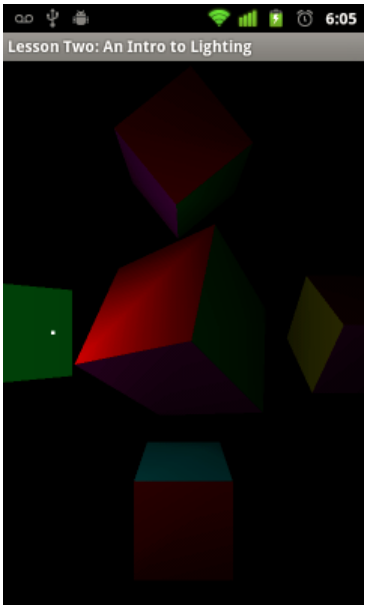
Learn OpenGL ES
Learn how to develop mobile graphics using OpenGL ES 2

Android Lesson Two: Ambient and Diffuse Lighting



Welcome to the second tutorial for Android. In this lesson, we're going to learn how to implement Lambertian reflectance using shaders, otherwise known as your standard diffuse lighting. In OpenGL ES 2, we need to implement our own lighting algorithms, so we will learn how the math works and how we can apply it to our scenes.



ASSUMPTIONS AND PREREQUISITES

Each lesson in this series builds on the lesson before it. Before we begin, please review the first lesson as this lesson will build upon the concepts introduced there.

WHAT IS LIGHT?

A world without lighting would be a dim one, indeed. Without light, we would not even be able to perceive the world or the objects that lie around us, except via the other senses such as sound and touch. Light shows us how bright or dim something is, how near or far it is, and what angle it lies at.

In the real world, what we perceive as light is really the aggregation of trillions of tiny particles called photons, which fly out of a light source, bounce around thousands or millions of times, and eventually reach our eye where we perceive it as light.

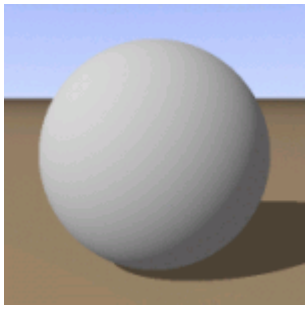
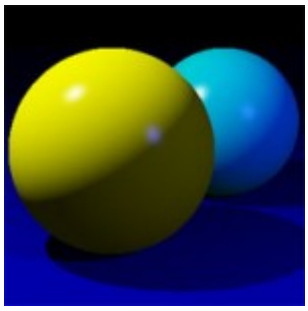
How can we simulate the effects of light via computer graphics? There are two popular ways to do it: ray tracing, and rasterisation. Ray tracing works by mathematically tracing actual rays of light and seeing where they end up. This technique gives very accurate and realistic results, but the downside is that simulating all of those rays is very computationally expensive, and usually too slow for real-time rendering. Due to this limitation, most real-time computer graphics use rasterisation instead, which simulates lighting by approximating the result. Given the realism of recent games, rasterisation can also look very nice, and is fast enough for real-time graphics even on mobile phones. Open GL ES is primarily a rasterisation library, so this is the approach we will focus on.

The different kinds of light

It turns out that we can abstract the way that light works and come up with three basic types of lighting:

Ambient lighting.

Ambient lighting
This is a base level of lighting that seems to pervade an entire scene. It is light that doesn't appear to come from any light source in particular because it has bounced around so much before reaching you. This type of lighting can be experienced outdoors on an overcast day, or indoors as the cumulative effect of

	many different light sources. Instead of calculating all of the individual lights, we can just set a base light level for the object or scene.
 *An example of ambient and diffuse lighting.*	**Diffuse lighting** This is light that reaches your eye after bouncing directly off of an object. The illumination level of the object varies with its angle to the lighting. Something facing the light head on is lit more brightly than something facing the light at an angle. Also, we perceive the object to be the same brightness no matter which angle we are at relative to the object. This is otherwise known as Lambert's cosine law. Diffuse lighting or Lambertian reflectance is common in everyday life and can be easily seen on a white wall lit up by an indoor light.
 *An example of specular highlights.*	**Specular lighting** Unlike diffuse lighting, specular lighting changes as we move relative to the object. This gives "shininess" to the object and can be seen on "smoother" surfaces such as glass and other shiny objects.

SIMULATING LIGHT

Just as there are three main types of light in a 3D scene, there are also three main types of light sources: directional, point, and spotlight. These can also be

easily seen in everyday life.

 *A brightly lit landscape.*	**Directional lighting** Directional lighting usually comes from a bright source that is so far away that it lights up the entire scene evenly and to the same brightness. This light source is the simplest type as the light is the same strength and direction no matter where you are in the scene.
 *An example of point lights.*	**Point lighting** Point lights can be added to a scene in order to give more varied and realistic lighting. The illumination of a point light falls off with distance, and its light rays travel out in all directions with the point light at the center.
 *Spotlight.*	**Spot lighting** In addition to the properties of a point light, spot lights also have the direction of light attenuated, usually in the shape of a cone.

The math

In this lesson, we're going to be looking at ambient lighting and diffuse lighting coming from a point source.

Ambient lighting

Ambient lighting is really indirect diffuse lighting, but it can also be thought of as a low-level light which pervades the entire scene. If we think of it that way, then it becomes very easy to calculate:

```
final color = material color * ambient light color
```

For example, let's say our object is red and our ambient light is a dim white. Let's assume that we store color as an array of three colors: red, green, and blue, using the RGB color model:

```
final color = {1, 0, 0} * {0.1, 0.1, 0.1} = {0.1, 0.0, 0.0}
```

The final color of the object will be a dim red, which is what you'd expect if you had a red object illuminated by a dim white light. There is really nothing more to basic ambient lighting than that, unless you want to get into more advanced lighting techniques such as radiosity.

Diffuse lighting – point light source

For diffuse lighting, we need to add attenuation and a light position. The light position will be used to calculate the angle between the light and the surface, which will affect the surface's overall level of lighting. It will also be used to calculate the distance between the light and the surface, which determines the strength of the light at that point.

Step 1: Calculate the lambert factor.

The first major calculation we need to make is to figure out the angle between the surface and the light. A surface which is facing the light straight-on should be illuminated at full strength, while a surface which is slanted should get less illumination. The proper way to calculate this is by using Lambert's cosine law. If we have two vectors, one being from the light to a point on the surface, and the second being a surface normal (if the surface is a flat plane, then the surface normal is a vector pointing straight up, or orthogonal to that surface), then we can calculate the cosine by first normalizing each vector so that it has a length

of one, and then by calculating the dot product of the two vectors. This is an operation that can easily be done via OpenGL ES 2 shaders.

Let's call this the **lambert factor**, and it will have a range of between **0** and **1**.

```
light vector = light position - object position
cosine = dot product(object normal, normalize(light vector))
lambert factor = max(cosine, 0)
```

Fitst we calculate the light vector by subtracting the object position from the light position. Then we get the cosine by doing a dot product between the object normal and the light vector. We normalize the light vector, which means to change its length so it has a length of one. The object normal should already have a length of one. Taking the dot product of two normalized vectors gives you the cosine between them. Because the dot product can have a range of **-1** to **1**, we clamp it to a range of **0** to **1**.

Here's an example with an flat plane at the origin and the surface normal pointing straight up toward the sky. The light is positioned at {0, 10, -10}, or 10 units up and 10 units straight ahead. We want to calculate the light at the origin.

```
light vector = {0, 10, -10} - {0, 0, 0} = {0, 10, -10}
object normal = {0, 1, 0}
```

In plain English, if we move from where we are along the light vector, we reach the position of the light. To normalize the vector, we divide each component by the vector length:

```
light vector length = square root(0*0 + 10*10 + -10*-10) = square root(200
normalized light vector = {0, 10/14.14, -10/14.14} = {0, 0.707, -0.707}
```

Then we calculate the dot product:

```
dot product({0, 1, 0}, {0, 0.707, -0.707}) = (0 * 0) + (1 * 0.707) + (0 *
```

Here is a good explanation of the dot product and what it calculates. Finally, we clamp the range:

```
lambert factor = max(0.707, 0) = 0.707
```

OpenGL ES 2's shading language has built in support for some of these functions so we don't need to do all of the math by hand, but it can still be useful to understand what is going on.

Step 2: Calculate the attenuation factor.

Next, we need to calculate the attenuation. Real light attenuation from a point light source follows the inverse square law, which can also be stated as:

```
luminosity = 1 / (distance * distance)
```

Going back to our example, since we have a distance of 14.14, here is what our final luminosity looks like:

```
luminosity = 1 / (14.14*14.14) = 1 / 200 = 0.005
```

As you can see, the inverse square law can lead to a strong attenuation over distance. This is how light from a point light source works in the real world, but since our graphics displays have a limited range, it can be useful to dampen this attenuation factor so we still get realistic lighting without things looking too dark.

Step 3: Calculate the final color.

Now that we have both the cosine and the attenuation, we can calculate our final illumination level:

```
final color = material color * (light color * lambert factor * luminosity)
```

Going with our previous example of a red material and a full white light source, here is the final calculation:

```
final color = {1, 0, 0} * ({1, 1, 1} * 0.707 * 0.005}) = {1, 0, 0} * {0.00
```

To recap, for diffuse lighting we need to use the angle between the surface and the light as well as the distance between the surface and the light in order to calculate the final overall diffuse illumination level. Here are the steps:

```
//Step one
light vector = light position - object position
```

```
cosine = dot product(object normal, normalize(light vector))
lambert factor = max(cosine, 0)

//Step two
luminosity = 1 / (distance * distance)

//Step three
final color = material color * (light color * lambert factor * luminosity)
```

Putting this all into OpenGL ES 2 shaders

The vertex shader

```
  final String vertexShader =
    "uniform mat4 u_MVPMatrix;      \n"     // A constant representing the
  + "uniform mat4 u_MVMatrix;       \n"     // A constant representing the
  + "uniform vec3 u_LightPos;       \n"     // The position of the light i

  + "attribute vec4 a_Position;     \n"     // Per-vertex position informa
  + "attribute vec4 a_Color;        \n"     // Per-vertex color informatio
  + "attribute vec3 a_Normal;       \n"     // Per-vertex normal informati

  + "varying vec4 v_Color;          \n"     // This will be passed into th

  + "void main()                    \n"     // The entry point for our ver
  + "{                              \n"
// Transform the vertex into eye space.
  + "   vec3 modelViewVertex = vec3(u_MVMatrix * a_Position);
// Transform the normal's orientation into eye space.
  + "   vec3 modelViewNormal = vec3(u_MVMatrix * vec4(a_Normal, 0.0));
// Will be used for attenuation.
  + "   float distance = length(u_LightPos - modelViewVertex);
// Get a lighting direction vector from the light to the vertex.
  + "   vec3 lightVector = normalize(u_LightPos - modelViewVertex);
// Calculate the dot product of the light vector and vertex normal. If the
// pointing in the same direction then it will get max illumination.
  + "   float diffuse = max(dot(modelViewNormal, lightVector), 0.1);
// Attenuate the light based on distance.
  + "   diffuse = diffuse * (1.0 / (1.0 + (0.25 * distance * distance)));
// Multiply the color by the illumination level. It will be interpolated a
  + "   v_Color = a_Color * diffuse;
// gl_Position is a special variable used to store the final position.
// Multiply the vertex by the matrix to get the final point in normalized
  + "   gl_Position = u_MVPMatrix * a_Position;
  + "}
```

There is quite a bit going on here. We have our combined model/view/projection matrix as in lesson one, but we've also added a model/view matrix. Why? We will need this matrix in order to calculate the distance between the position of the light source and the position of the current vertex. For diffuse lighting, it

actually doesn't matter whether you use world space (model matrix) or eye space (model/view matrix) so long as you can calculate the proper distances and angles.

We pass in the vertex color and position information, as well as the surface normal. We will pass the final color to the fragment shader, which will interpolate it between the vertices. This is also known as Gouraud shading.

Let's look at each part of the shader to see what's going on:

```
// Transform the vertex into eye space.
  + "   vec3 modelViewVertex = vec3(u_MVMatrix * a_Position);
```

Since we're passing in the position of the light in eye space, we convert the current vertex position to a coordinate in eye space so we can calculate the proper distances and angles.

```
// Transform the normal's orientation into eye space.
  + "   vec3 modelViewNormal = vec3(u_MVMatrix * vec4(a_Normal, 0.0));
```

We also need to transform the normal's orientation. Here we are just doing a regular matrix multiplication like with the position, but if the model or view matrices have been scaled or skewed, this won't work: we'll actually have to undo the effect of the skew or scale by multiplying the normal by the transpose of the inverse of the original matrix. This website best explains why we have to do this.

```
// Will be used for attenuation.
  + "   float distance = length(u_LightPos - modelViewVertex);
```

As shown before in the math section, we need the distance in order to calculate the attenuation factor.

```
// Get a lighting direction vector from the light to the vertex.
  + "   vec3 lightVector = normalize(u_LightPos - modelViewVertex);
```

We also need the light vector to calculate the Lambertian reflectance factor.

```
// Calculate the dot product of the light vector and vertex normal. If the
// pointing in the same direction then it will get max illumination.
```

```
  + "   float diffuse = max(dot(modelViewNormal, lightVector), 0.1);
```

This is the same math as above in the math section, just done in an OpenGL ES 2 shader. The 0.1 at the end is just a really cheap way of doing ambient lighting (the value will be clamped to a minimum of 0.1).

```
// Attenuate the light based on distance.
  + "   diffuse = diffuse * (1.0 / (1.0 + (0.25 * distance * distance)));
```

The attenuation math is a bit different than above in the math section. We scale the square of the distance by 0.25 to dampen the attenuation effect, and we also add 1.0 to the modified distance so that we don't get oversaturation when the light is very close to an object (otherwise, when the distance is less than one, this equation will actually brighten the light instead of attenuating it).

```
// Multiply the color by the illumination level. It will be interpolated a
  + "   v_Color = a_Color * diffuse;
// gl_Position is a special variable used to store the final position.
// Multiply the vertex by the matrix to get the final point in normalized
  + "   gl_Position = u_MVPMatrix * a_Position;
```

Once we have our final light color, we multiply it by the vertex color to get the final output color, and then we project the position of this vertex to the screen.

The pixel shader

```
final String fragmentShader =
  "precision mediump float;       \n"     // Set the default precision to
                                          // precision in the fragment sha
+ "varying vec4 v_Color;          \n"     // This is the color from the ve
                                          // triangle per fragment.
+ "void main()                    \n"     // The entry point for our fragr
+ "{                              \n"
+ "   gl_FragColor = v_Color;     \n"     // Pass the color directly throu
+ "}                              \n";
```

Because we are calculating light on a per-vertex basis, our fragment shader looks the same as it did in the first lesson — all we do is pass through the color directly through. In the next lesson, we'll look at per-pixel lighting.

PER-VERTEX VERSUS PER-PIXEL LIGHTING

In this lesson we have focused on implementing per-vertex lighting. For diffuse lighting of objects with smooth surfaces, such as terrain, or for objects with many triangles, this will often be good enough. However, when your objects don't contain many vertices (such as our cubes in this example program) or have sharp corners, vertex lighting can result in artifacts as the light level is linearly interpolated across the polygon; these artifacts also become much more apparent when specular highlights are added to the image. More can be seen at the Wiki article on Gouraud shading.

AN EXPLANATION OF THE CHANGES TO THE PROGRAM

Besides the addition of per-vertex lighting, there are other changes to the program. We've switched from displaying a few triangles to a few cubes, and we've also added utility functions to load in the shader programs. There are also new shaders to display the position of the light as a point, as well as other various small changes.

Construction of the cube

In lesson one, we packed both position and color attributes into the same array, but OpenGL ES 2 also lets us specify these attributes in seperate arrays:

```
// X, Y, Z
final float[] cubePositionData =
{
        // In OpenGL counter-clockwise winding is default. This means that
        // if the points are counter-clockwise we are looking at the "fron
        // the back. OpenGL has an optimization where all back-facing tria
        // usually represent the backside of an object and aren't visible

        // Front face
        -1.0f, 1.0f, 1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        -1.0f, -1.0f, 1.0f,
        1.0f, -1.0f, 1.0f,
        1.0f, 1.0f, 1.0f,
        ...

// R, G, B, A
final float[] cubeColorData =
{
        // Front face (red)
        1.0f, 0.0f, 0.0f, 1.0f,
```

```
        1.0f, 0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
        1.0f, 0.0f, 0.0f, 1.0f,
        ...
```

New OpenGL flags

We have also enabled culling and the depth buffer via glEnable() calls:

```
// Use culling to remove back faces.
GLES20.glEnable(GLES20.GL_CULL_FACE);

// Enable depth testing
GLES20.glEnable(GLES20.GL_DEPTH_TEST);
```

As an optimization, you can tell OpenGL to eliminate triangles that are on the back side of an object. When we defined our cube, we also defined the three points of each triangle so that they are counter-clockwise when looking at the "front" side. When we flip the triangle around so we're looking at the "back" side, the points then appear clockwise. You can only ever see three sides of a cube at the same time so this optimization tells OpenGL to not waste its time drawing the back sides of triangles.

Later when we draw transparent objects we may want to turn culling back off, as then it will be possible to see the back sides of objects.

We've also enabled depth testing. If you always draw things in order from back to front then depth testing is not strictly necessary, but by enabling it not only do you not need to worry about the draw order (although rendering can be faster if you draw closer objects first), but some graphics cards will also make optimizations which can speed up rendering by spending less time drawing pixels that will be drawn over anyways.

Changes in loading shader programs

Because the steps to loading shader programs in OpenGL are mostly the same, these steps can easily be refactored into a separate method. We've also added the following calls to retrieve debug info, in case the compilation/link fails:

```
GLES20.glGetProgramInfoLog(programHandle);
GLES20.glGetShaderInfoLog(shaderHandle);
```

Vertex and shader program for the light point

There is a new vertex and shader program specifically for drawing the point on the screen that represents the current position of the light:

```
// Define a simple shader program for our point.
final String pointVertexShader =
    "uniform mat4 u_MVPMatrix;      \n"
  + "attribute vec4 a_Position;     \n"
  + "void main()                    \n"
  + "{                              \n"
  + "   gl_Position = u_MVPMatrix   \n"
  + "               * a_Position;   \n"
  + "   gl_PointSize = 5.0;         \n"
  + "}                              \n";

final String pointFragmentShader =
    "precision mediump float;       \n"
  + "void main()                    \n"
  + "{                              \n"
  + "   gl_FragColor = vec4(1.0,    \n"
  + "   1.0, 1.0, 1.0);             \n"
  + "}                              \n";
```

This shader is similar to the simple shader from the first lesson. There's a new property, gl_PointSize which we hard-code to 5.0; this is the output point size in pixels. It's used when we draw the point using GLES20.GL_POINTS as the mode. We've also hard-coded the output color to white.

FURTHER EXERCISES

- Try removing the "oversaturation protection" and see what happens.
- There is a flaw with the way the lighting is done. Can you spot what it is? Hint: What is the downside of the way I did ambient lighting, and what happens to the alpha?
- What happens if you add a gl_PointSize to the cube shader and draw it using GL_POINTS?

FURTHER READING

- Fallout Software: OpenGL Lighting or How Light Sources Work (Long, In-depth Tutorial)
- Clockworkcoders Tutorials: Per Fragment Lighting
- Lighthouse3D.com: The Normal Matrix
- arcsynthesis.org: OpenGL Tutorials: Normal Transformation
- OpenGL Programming Guide: Chapter 5 Lighting

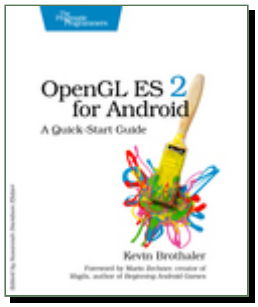
The further reading section above was an invaluable resource to me while writing this tutorial, so I highly recommend reading them for more information and explanations.

WRAPPING UP

The full source code for this lesson can be downloaded from the project site on GitHub.

A compiled version of the lesson can also be downloaded directly from the Android Market:

Thanks for getting through another big lesson! I learned a lot while writing it, and I hope you learned a lot by following it through as well. Feel free to ask any questions or offer feedback, and thanks for stopping by!



Buy Now

ABOUT THE BOOK

Android is booming like never before, with millions of devices shipping every day. In *OpenGL ES 2 for Android: A Quick-Start Guide*, you'll learn all about shaders and the OpenGL pipeline, and discover the power of OpenGL ES 2.0, which is much more feature-rich than its predecessor.

It's never been a better time to learn how to create your own 3D games and live wallpapers. If you can program in Java and you have a creative vision that you'd like to share with the world, then this is the book for you.

Share / Save ...

Author: Admin

Kevin is the author of OpenGL ES 2 for Android: A Quick-Start Guide. He also has extensive experience in Android development. View all posts by Admin

Admin / June 15, 2011 / Android, Android Tutorials / ambient lighting, attenuation, diffuse lighting, Gouraud shading, Johann Heinrich Lambert, Lambertian reflectance, lighting, per-vertex lighting, quadratic attenuation

84 thoughts on "Android Lesson Two: Ambient and Diffuse Lighting"

Ryan Chavez

June 17, 2011 at 9:48 pm

This is the first example that I have found that actually had a good description on how to get lighting working properly. Thank you!

Admin

June 23, 2011 at 2:05 am

Thanks for the feedback, Ryan, I appreciate it! I hope to add some more explanations behind how the lighting works and why we use the math we do.

Pingback: Android Lesson Three: Moving to Per-Fragment Lighting | Learn OpenGL ES

Alex

October 7, 2011 at 12:58 am

These tutorials are great! they describe every step in detail without assuming that your a mathematician. thanks for this!

RuiLang

November 10, 2011 at 7:45 am

我不得不说这是我看过的最好的编程指南，详细而且全面。非常感谢国外的学者发表这么出色的指南！
我将会把所有的这些学习指南研读N遍！
对你附加的其他学习资料也表示真心的感谢！
非常感谢你的无私！

I have to say this is the best programming guides that I have seen on the net.
I will put all of these study guides read N times!
Other learning materials attached to your tutorials is also very useful for me !

Thank you very much!

Admin

November 14, 2011 at 9:33 pm

Thank you for the great comments and feedback! Within three weeks more articles will hopefully be going live. Sorry to keep everyone waiting. 🙂

Superfly

November 21, 2011 at 4:00 pm

Not only is this one of the very few OGL2.0 android tuts on the net, its a damn GOOD one at that! Especially the math parts with links to detailed information on the principles behind it, makes it understandable even for a dumbass like me! THANKS!!!!

infinity

January 4, 2012 at 11:14 pm

Our universe is described as infinitely large and the atomic universe is infinitely tiny. But what is infinitely massive and little in truly mathematical terms?

DaMachine

February 4, 2012 at 7:43 am

Thanks for this great series of tutorials, I've learned a lot. What is the answer to the question in the execise part? I assume the flaw is that you can't do seperate ambient color and diffuse color that way.

Go on with the excellent work 😉

Neil Kolban

March 8, 2012 at 5:57 am

Thank you SO much for these tutorials. Please keep them coming.

Admin

March 9, 2012 at 3:24 pm

Thanks everyone again for the great feedback. As for the flaw with the ambient lighting, it's been so long since I wrote this tutorial, but looking at it again, it looks like the flaw is that the ambient lighting is still attenuated with distance. Maybe that's not a flaw depending on the effect you're going for. 🙂

Alvin Then

April 2, 2012 at 9:24 am

Thank you very much for the tutorials provided over here. It's really helping me a total noob to venture into 3D world 🙂
One question, what is generic vertex attributes array? When should we enable/disable them?

Thanks!

Admin

April 3, 2012 at 12:27 am

Hi Alvin,

Thanks for the compliments! In OpenGL ES 2.0, there are no longer fixed attributes for specific features, such as color, normal, etc.... so you use generic attributes for these. The meaning of these attributes is now interpreted in the shaders. In this lesson for example, we use generic vertex attributes to represent our position and color. You also need to call glEnableVertexAttribArray before you can pass the vertex data through to a shader. It seems that you only need to call this once and not on every draw frame.

Hope this helps!

Alvin Then

April 3, 2012 at 9:50 am

Yup that helped me to understand more. Thanks again! 🙂

Pingback: Android Lesson Four: Introducing Basic Texturing | Learn OpenGL ES

---

zqWu

April 22, 2012 at 4:58 am

i have learnt lesson one ,it's pretty good!

Admin

April 22, 2012 at 8:16 pm

Thanks! 🙂

---

zqWu

April 22, 2012 at 6:13 am

i seperate the Class into several class, a class, aclass,but keep all the matrix and handle, and the program is ok to run, 5 cubes and a point light. but when i compare to the original effect, i found the light point failed to light the cube

any suggestion?

Admin

April 22, 2012 at 8:16 pm

My guess is that there's an error with either the shader or the input values. Maybe try from the original source and make your modifications one by one?

---

zqWu

April 23, 2012 at 2:52 am

i get the error,
mLightPosInModelSpace = new float[4];
but in original code its
float[] mLightPosInModelSpace = {0.0f, 0.0f, 0.0f, 1.0f};

zqWu

April 22, 2012 at 6:20 am

second question:in your code i found something like this

public static final String pointFragmentShader =
“precision mediump float; \n”
+ “void main() \n”
+ “{ \n”
+ ” gl_FragColor = vec4(1.0, \n”
+ ” 1.0, 1.0, 1.0); \n”
+ “} \n”;
is there a way to get the GLSL code out of the String and reach the same effect? or is there another way i can create a shader?

Admin

April 22, 2012 at 8:15 pm

Yes, you can place them as text files under res/raw, and read them in using the code found here: https://github.com/learnopengles/Learn-OpenGLES-Tutorials/blob/master/android/AndroidOpenGLESLessons/src/com/learnopengles/android/common/RawResourceReader.java

There should be an example of that in the last couple of lessons at the source code. Let me know if that helps!

steve

April 22, 2012 at 5:24 pm

Maybe you should mention more about gl_PointSize.

For example I found out you need to set glEnable(GL_POINT_SMOOTH) if you want to make the point size larger than 1 pixel. At least on my desktop

implementation. My maximum point size is 63 and you can find out yours using this code:

```
GLfloat pointSizeRange[2];
glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, pointSizeRange);

GLfloat min = pointSizeRange[0];
GLfloat max = pointSizeRange[1];
```

Admin

April 22, 2012 at 8:14 pm

Thanks for sharing that, Steve!

Sam

May 1, 2012 at 4:50 pm

Another really easy way is to store the shader code in a string in res/strings and then import it in the activity with getResources().getString(R.string.yourshadercode); 😀

Admin

May 2, 2012 at 1:05 pm

Hi Sam,

True, this could be even easier than the raw resource. I haven't tried to see what happens to the formatting though with multi-line; does it get preserved decently well?

edie

May 10, 2012 at 4:22 am

awesome tutorial,
thank you !

Pingback: Android Lesson Eight: An Introduction to Index Buffer Objects (IBOs) | Learn OpenGL ES

umanga

May 17, 2012 at 9:34 am

Great set of tutorials.
Finally managed to make my first OpenGLES2 app running thanks to your tutorials.

Admin

May 29, 2012 at 6:10 am

Sweet! 🙂 Glad to hear it.

Joey

June 29, 2012 at 9:52 pm

Hi, great tutorial! Just one question... what value do you pass into a_Normal?

Admin

July 1, 2012 at 5:44 pm

Hi Joey,

If you imagine that the current vertex is part of a flat plane, representing the underlying surface, then the a_Normal should be a vector pointing directly away from that plane. Let me know if this illustration from Wolfram helps out. 🙂

Joey

July 2, 2012 at 2:06 am

Hi Admin,

Thanks for the reply... I ended up figuring it out, but then I ran into a different problem with directional light, I sent you an email (from contact form), hope that's okay

Martin

August 19, 2012 at 6:37 pm

How can I calculate specular lighting. I simply need a term

Admin

August 21, 2012 at 2:01 am

This looks like a good resource:
http://www.opengl.org/sdk/docs/tutorials/ClockworkCoders/lighting.php

TranThuan

October 11, 2012 at 6:48 am

thanks so much. these series of tutorial are great. thanks again!

Admin

October 12, 2012 at 1:07 am

No problem 🙂

Buzeeg

October 20, 2012 at 10:30 pm

What I did to fix the ambiant light which was depending on the distance :

```
uniform mat4 u_MVPMatrix;
uniform mat4 u_MVMatrix;
uniform vec3 u_LightPos;
attribute vec4 a_Position;
attribute vec4 a_Color;
attribute vec3 a_Normal;
varying vec4 v_Color;
void main()
{
vec3 modelViewVertex = vec3(u_MVMatrix * a_Position);
vec3 modelViewNormal = vec3(u_MVMatrix * vec4(a_Normal, 0.0));
float distance = length(u_LightPos - modelViewVertex);
vec3 lightVector = normalize(u_LightPos - modelViewVertex);
float diffuse = max(dot(modelViewNormal, lightVector), 0.0); // remove approx
ambient light
diffuse = diffuse * (1.0 / (1.0 + (0.25 * distance * distance)));
v_Color = a_Color * (diffuse + 0.3)/2.0; // average between ambient and diffuse
gl_Position = u_MVPMatrix * a_Position;
}
```

I hope this is correct 🙂

Admin

October 26, 2012 at 9:00 pm

Well, as they say there's more than one way. 😉 I don't see anything necessarily wrong with this! How does it look?

FromHongKong

November 13, 2012 at 5:19 pm

This tutorial is of very high quality. It pinpoint the essentials of how the user program in android interact with , and how tasks can be off-loaded to shaders, in a clear and detailed way.
I am making my first homebrew game apps with OpenGL ES2 for learning purpose , these tutorials serve as an excellent starting point for my project.

Thank you so much.

Admin

November 20, 2012 at 2:54 am

Thanks for the great compliments, I really appreciate it and I'd love to see what you come up with and would be happy to help promote as well! 🙂

FipS

November 26, 2012 at 8:03 pm

It's a very clear tutorial, congratulations! I especially like your "Different kinds of light" introduction, it's very useful, thanks for your effort. FipS

Pingback: CS 491 Lecture 21 – OpenGL | teaching machines

paul

December 27, 2012 at 1:06 am

Hi there,

first off thanks for the tutorial, you really did a great job here.

The one question I have is where we actually “move” our point or set the moving path for it.
At the moment my guess would be in the drawLight method at GLES20.glVertexAttrib3f()?

Maybe i just have a blackout but could someone give me a hint on this please?

Thanks in advance

Admin

December 27, 2012 at 8:45 pm

You’re on the right track, we draw the actual point in these lines: https://github.com/learnopengles/Learn-OpenGLES-Tutorials/blob/master/android/AndroidOpenGLESLessons/src/com/learnopengles/android/lesson2/LessonTwoRenderer.java#L532-L550

It’s animated at these lines: https://github.com/learnopengles/Learn-OpenGLES-Tutorials/blob/master/android/AndroidOpenGLESLessons/src/com/learnopengles/android/lesson2/LessonTwoRenderer.java#L443-L450

I need to update all of these tutorials to make the source code clearer; it’s been a while. 🙂

Deduu

January 28, 2013 at 4:30 pm

Hi, Thanks for writing these tutorial. But there was something that’s not clear to me.
Here, you said

//Step three
final color = material color * (light color * lambert factor * luminosity)

But in vertex shader, I did not find the light color to be multiplied also.

// Multiply the color by the illumination level. It will be interpolated across the triangle.
+ ” v_Color = a_Color * diffuse;

Is there something missed in my observation?

thanks

Admin

January 30, 2013 at 3:06 pm

In this case, the light color is assumed to be white (it’s been a while since I touched the code, but this is what it looks like to me).

Wojtek

February 14, 2013 at 9:59 am

Hi there.
I want to separate some fragments of scene so I’ve made a few classes. I’ve got a class Triangle which takes class Shaders as parameter and describes triangles in the same color (lighted by one shader). Class Shders describes shaders and takes string codes for shaders as a parameter.
All other parameters are just like code from this lessons.
My object’s color is blue, but when I add light, it’s color changes to red. Why is that? Does anyone have some ideas? I can share my code by e-mail. Contact me if You think You could help. I will be so glad.

Admin

February 16, 2013 at 4:17 pm

I recommend posting a question to StackOverflow with source attached, and you can always share the link here. Sounds like it has something to do with

the way the color is multiplied or assigned to gl_FragColor.

Wojtek

February 17, 2013 at 9:09 am

Ok. Thank You for answer. So I put my code there:

http://wklej.org/hash/f860ebe2c35/
http://wklej.org/hash/86db129dd62/
http://wklej.org/hash/0e1be3e5cff/

If anybody could help feel free to mail me: lesniakwojciech at gmail dot com or post a replay if not too long.

El Gringo

April 3, 2013 at 2:31 pm

Thank you very much for these wonderfull tutorials. It has cost me about a week to get through lesson one and two including the necessary mathematical sidetrips. But it was worth it. 😉

One question. You are using triangles to build the faces of the cube. Isn't there a way to use Quads in Gles2?

Iwas thinking about an OBJ importer. And it would be much easier if wouldn't have to convert everything to triangles.

Admin

April 3, 2013 at 3:39 pm

There is no GL_QUADS in ES, so you'd need to break each one down into two triangles. I recall a couple others here were working on an OBJ importer — I have to see where they're at. Please feel free to post something in the forums as well if you ever want to share some code there. 🙂

Shubham K

June 10, 2013 at 2:18 pm

the code has cubeNormalData array, i don't find it explained in the tutorial.. where are we using it?

Admin

June 10, 2013 at 3:35 pm

Good point, the normals themselves were elided to keep the size of the tutorial down. These normals are the source for the normals that we talk about in the tutorial, so when the shaders run those normals will be used for the lighting. Let me know if that helps out a bit.

rohit

July 5, 2013 at 8:30 am

Hi,

Your Articles are really good. But i tried implementing a pyramid using the same code. I get nothing except the black scrren. Are there any debuggers for opengl that you are aware of. I searched on the web only thing I found was a tracer for android 4.1 . But I have android 4.04 . I am finding it really difficult to debug the code. Ay help is much appreciated.

Thanks

Admin

July 8, 2013 at 10:10 pm

The tracer unfortunately doesn't work very well even if you have a supported OS. Old-school techniques work better: you can try adding calls to glGetError

and also step through the code, and also try turning off depth testing and face culling if you have those enabled. Turn off texturing if you're doing that, eventually by simplifying things enough you'll likely find the root cause.

If you're lucky you might also be able to use one of the GPU-specific tools without too much trouble, NVIDIA has some and the other GPU vendors do as well.

rohit

July 24, 2013 at 6:57 am

thanks 🙂

štěty

July 25, 2013 at 5:01 pm

Can someone please tell me the answer to this question is that in Further Exercises:
There is a flaw with the way the ambient lighting is done. Can you spot what it is?

Admin

July 25, 2013 at 7:30 pm

The ambient should probably be done like this:

float diffuse = max(dot(modelViewNormal, lightVector), 0.0);
...
v_Color = 0.1 + ...

Instead of:

float diffuse = max(dot(modelViewNormal, lightVector), 0.1);

štěty

July 26, 2013 at 4:40 pm

And look how the entire line v_Color?
Like this: v_Color = 0.1 + a_Color * diffuse;?
Because if I do this so I have all the colors faded.
So how is it right?

Admin

July 27, 2013 at 11:34 am

Yeah, that's what ambient lighting does. 😉 It's essentially a hack to simulate global illumination. In a scene like this you probably don't want too much of it.

štěty

July 30, 2013 at 9:14 am

Thank you very much.

štěty

August 8, 2013 at 6:33 pm

How can I add 2,3 ... light?
I have to create all the variables with the light and make it twice as 2x normal light. Or is used in a shader trick.
Best would be to show a sequence of code for understanding.
(It may also be per-fragment lighting.)
Thank you for any answers 🙂

Admin

August 9, 2013 at 2:57 am

> Multiple lights works the same as for one light, you just add the contribution from each light together. One way you can do it is by passing in a uniform array for the lights, and just loop over that array and add the contribution from each light. There's an example of this in the source code from my book, available here: http://pragprog.com/titles/kbogla/source_code. The example is with different types of light but should still help you get an idea of how to do it; just check out the code for "Lighting".

štěty

August 14, 2013 at 8:14 am

Thank you. Early look at it.

Sam

January 28, 2014 at 2:49 pm

Hi,

thanks for the tutorials… i really appreciate the help in learning…

but I'm having a little bit of a problem… for some reason the bottom face of my cubes is transparent… only the bottom face… at first i thought maybe it had to do with the normal, eye level, lighting position, ect… so i adjusted each of these in turn to see what effect it would have… nothing…

I reproduced the code as closely as i could for tutorials one and two, and i ran through the code on github to see if i could find my error… as far as i can see it is very nearly identical, with the exception of the getshader methods…

can you possibly point me at where i might be making an error?

Christian

April 24, 2014 at 4:02 pm

Hi, good Tutorial!
How can I define the size or intensity of the light source?

Admin

April 29, 2014 at 10:05 pm

Hi Christian, one way you could do that could be to change the colour value of the light source (higher values will affect the intensity), or by changing the position & the falloff value. Since you have full control through the shader, you can experiment with different options. 🙂

Greg Sidelnikov

June 23, 2014 at 3:31 pm

Thank you for including my tutorial website on the sidebar of your website. I am currently working on writing new material that covers modern OpenGL techniques, that will make it to the site soon. I realized that the glBegin/glEnd/glVertex technique has become outdated over the years in favor of shaders. Nonetheless, OpenGL Light tutorial is still adequate for OpenGL lighting theory and covers different types of light and how they work. I have just obtained some new OpenGL books and I'm excited to start documenting the next series of tutorials covering shader-based GLSL lighting techniques.

Admin

June 23, 2014 at 8:53 pm

That would be really great! Please let me know if I need to update any links. And... if you have the chance, maybe you could also check out my book for Android. 😉

Paul

July 17, 2014 at 8:17 am

This tutorial is good but when i scroll down theres no output for the code. 😕 But thanks anyway for the tutorial. Now i know, lighting is just math.

Admin

July 29, 2014 at 12:49 pm

Hmm, it might be a JS issue. You can also access the full code here: https://github.com/learnopengles/Learn-OpenGLES-Tutorials

---

Paul

July 17, 2014 at 8:28 am

I have question. What if the light source is moving? Did this tutorial already point this question? Email me please. thanks.

Admin

July 29, 2014 at 12:49 pm

The light source is moving in this tutorial; try it out. 😉

Paul

July 18, 2014 at 4:05 pm

This example, good thought it is, applies light attenuation to the alpha value in v_Color. That'll makes dark things transparent if you're using the alpha channel. That caught me out for a while.

Strahlex

July 19, 2014 at 5:33 pm

Found the error: The alpha value of the color is also "diffused" meaning the objects gets transparent when moved away from the light source. To correct

this add this to the vertex shader:
v_Color[3] = a_Color[3]; // correct alpha

Fred

December 14, 2014 at 10:25 pm

Using the framework from this example, I tried creating different shapes but found strange effects. After a lot of trial and error, I discovered that this can most easily reproduced by removing all but one sides of the cube. In my example, I removed all faces except for the back face (blue). I would expect to see nothing when it is facing away from the eye or if the light is behind the face (from the eye's perspective). Instead I'm seeing streaks of various colors and occasionally I see the blue face.

Admin

January 5, 2015 at 4:15 pm

The lighting code wasn't written with two-sided lighting in mind, but you can check out http://www.learnopengles.com/android-lesson-eight-an-introduction-to-index-buffer-objects-ibos/ for an example of where this is done.

Lawrence D'Oliveiro

February 1, 2015 at 2:48 am

Don't you think it's a shame that Java doesn't allow implicit string concatenation? C and C++ allow it, even Python has it. It would save on a whole lot of plus signs.

Pingback: [FRAGE] Gouraud shading

Chuck

November 13, 2016 at 3:48 pm

Please check your lesson 2 webpage. I lose content in both Internet Explorer and Chrome as pages narrow and finally disappear.

Admin

November 16, 2016 at 9:46 pm

Thanks, I'm not sure what happened but the page source was mangled somehow. Should be fixed up now.

Matt

November 14, 2016 at 11:40 pm

Great tutorial, but the page has serious layout issues for me (chrome on Mac): https://www.dropbox.com/s/9un86bucawfs4k4/Screenshot%202016-11-14%2015.38.54.png?dl=0

Admin

November 16, 2016 at 9:46 pm

Sorry about that! It should be fixed now.