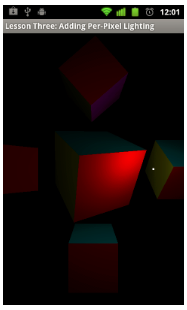
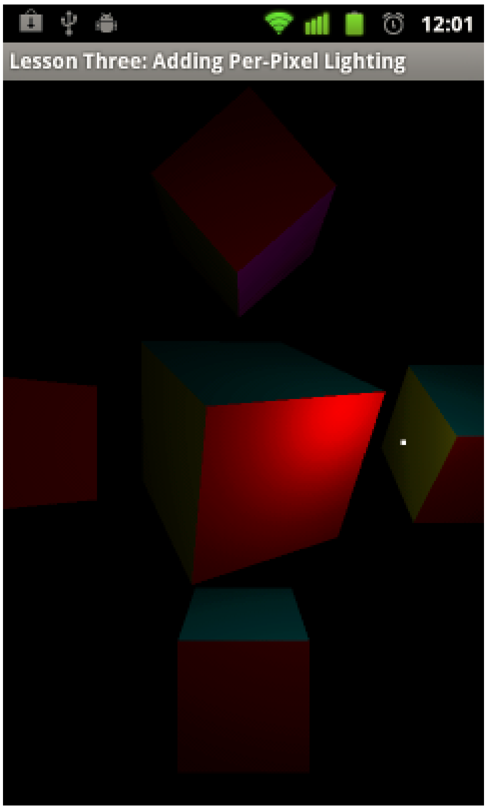
Learn OpenGL ES

Learn how to develop mobile graphics using OpenGL ES 2

Android Lesson Three: Moving to Per-Fragment Lighting



Welcome to the the third tutorial for Android! In this lesson, we're going to take everything we learned in lesson two and learn how to apply the same lighting technique on a per-pixel basis. We will be able to see the difference, even when using standard diffuse lighting with simple cubes.



Per fragment lighting; At the corner of a square.

ASSUMPTIONS AND PREREQUISITES

Each lesson in this series builds on the lesson before it. This lesson is an extension of lesson two, so please be sure to review that lesson before continuing on. Here are the previous lessons in the series:

- Android Lesson One: Getting Started
- Android Lesson Two: Ambient and Diffuse Lighting

WHAT IS PER-PIXEL LIGHTING?

Per-pixel lighting is a relatively new phenomenon in gaming with the advent of the use of shaders. Many famous old games such as the original Half Life were developed before the time of shaders and featured mainly static lighting, with some tricks for simulating dynamic lighting using either per-vertex (otherwise known as Gouraud shading) lights or other techniques, such as dynamic lightmaps.

Lightmaps can give a very nice result and can sometimes give even better results than shaders alone as expensive light calculations can be precomputed, but the downside is that they take up a lot of memory and doing dynamic lighting with them is limited to simple calculations.

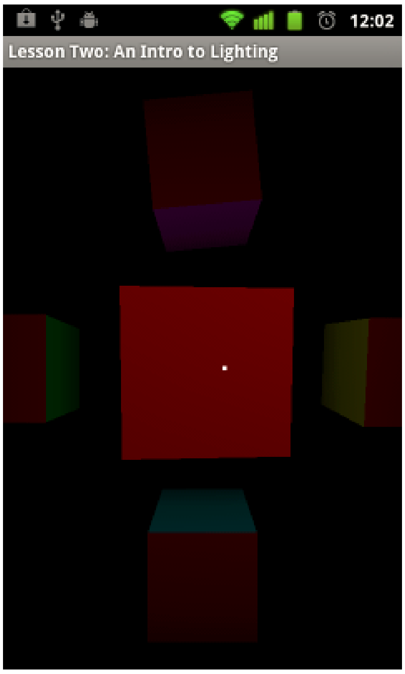
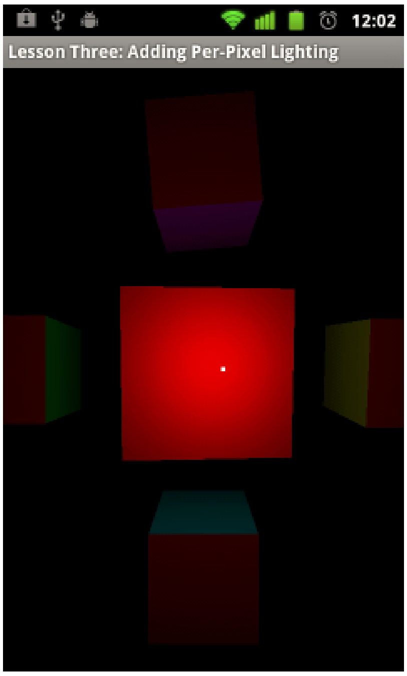
With shaders, a lot of these calculations can now be offloaded to the GPU, which allows for many more effects to be done in real-time.

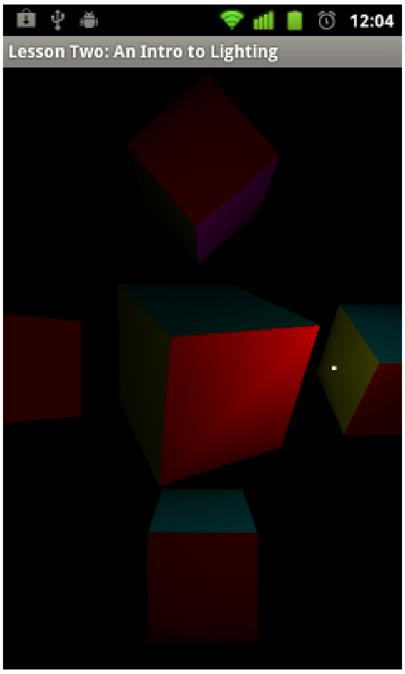
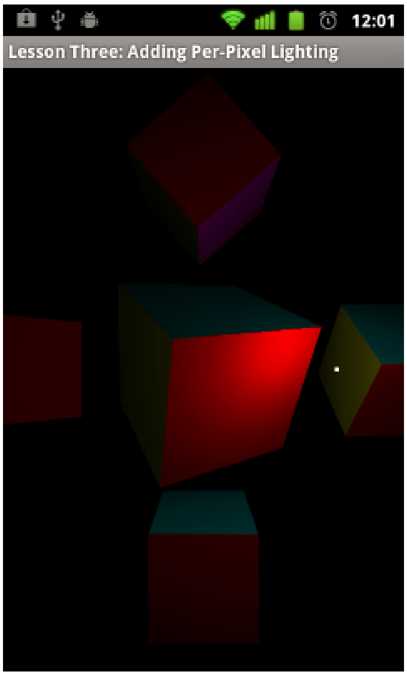
MOVING FROM PER-VERTEX TO PER-FRAGMENT LIGHTING

In this lesson, we're going to look at the same lighting code for a per-vertex solution and a per-fragment solution. Although I have referred to this type of lighting as per-pixel, in OpenGL ES we actually work with *fragments*, and several fragments can contribute to the final value of a pixel.

Mobile GPUs are getting faster and faster, but performance is still a concern. For "soft" lighting such as terrain, per-vertex lighting may be good enough. Ensure you have a proper balance between quality and speed.

A significant difference between the two types of lighting can be seen in certain situations. Take a look at the following screen captures:

 *Per vertex lighting; centered between four vertices of a square.*	 *Per fragment lighting; centered between four vertices of a square.*	In the per-vertex lighting in the left image, the front face of the cube appears as if flat-shaded, and there is no evidence of a light nearby. This is because each of the four points of the front face are more or less equidistant from the light, and the low light intensity at each of these four points is simply interpolated across the two triangles that make up the front face. The per-fragment version shows a nice highlight in comparison.
		The left image shows a [Gouraud-shaded](#) cube.As the light source moves near the corner of the front face of the cube, a triangle-like effect can be seen. This

 *Per vertex lighting; At the corner of a square.*	 *Per fragment lighting; At the corner of a square.*	is because the front face is actually composed of two triangles, and as the values are interpolated in different directions across each triangle we can see the underlying geometry. The per-fragment version shows no such interpolation errors and shows a nice circular highlight near the edge.

An overview of per-vertex lighting

Let’s take a look at our shaders from lesson two; a more detailed explanation on what the shaders do can be found in that lesson.

Vertex shader

```
uniform mat4 u_MVPMatrix;      // A constant representing the combined mode
uniform mat4 u_MVMatrix;       // A constant representing the combined mode
uniform vec3 u_LightPos;       // The position of the light in eye space.

attribute vec4 a_Position;     // Per-vertex position information we will p
attribute vec4 a_Color;        // Per-vertex color information we will pass
attribute vec3 a_Normal;       // Per-vertex normal information we will pas

varying vec4 v_Color;          // This will be passed into the fragment sha

// The entry point for our vertex shader.
void main()
{
    // 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
    // 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 interpolat
    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 normali
    gl_Position = u_MVPMatrix * a_Position;
}
```

Fragment shader

```
precision mediump float;       // Set the default precision to medium. We
                               // precision in the fragment shader.
varying vec4 v_Color;          // This is the color from the vertex shader
                               // triangle per fragment.

// The entry point for our fragment shader.
void main()
{
    gl_FragColor = v_Color;    // Pass the color directly through the pipe
}
```

As you can see, most of the work is being done in our vertex shader. Moving to per-fragment lighting means that our fragment shader is going to have more work to do.

Implementing per-fragment lighting

Here is what the code looks like after moving to per-fragment lighting.

Vertex shader

```
uniform mat4 u_MVPMatrix;      // A constant representing the combined mod
uniform mat4 u_MVMatrix;       // A constant representing the combined mod
```

```
attribute vec4 a_Position;     // Per-vertex position information we will
attribute vec4 a_Color;        // Per-vertex color information we will pas
attribute vec3 a_Normal;       // Per-vertex normal information we will pa

varying vec3 v_Position;       // This will be passed into the fragment sh
varying vec4 v_Color;          // This will be passed into the fragment sh
varying vec3 v_Normal;         // This will be passed into the fragment sh

// The entry point for our vertex shader.
void main()
{
    // Transform the vertex into eye space.
    v_Position = vec3(u_MVMatrix * a_Position);

    // Pass through the color.
    v_Color = a_Color;

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

    // gl_Position is a special variable used to store the final position.
    // Multiply the vertex by the matrix to get the final point in normali
    gl_Position = u_MVPMatrix * a_Position;
}
```

The vertex shader is simpler than before. We have added two linearly-interpolated variables to be passed through to the fragment shader: the vertex position and the vertex normal. Both of these will be used when calculating lighting in the fragment shader.

Fragment shader

```
precision mediump float;       // Set the default precision to medium. We
                               // precision in the fragment shader.
uniform vec3 u_LightPos;       // The position of the light in eye space.

varying vec3 v_Position;       // Interpolated position for this fragment.
varying vec4 v_Color;          // This is the color from the vertex shader
                               // triangle per fragment.
varying vec3 v_Normal;         // Interpolated normal for this fragment.

// The entry point for our fragment shader.
void main()
{
    // Will be used for attenuation.
    float distance = length(u_LightPos - v_Position);

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

    // Calculate the dot product of the light vector and vertex normal. If
    // pointing in the same direction then it will get max illumination.
    float diffuse = max(dot(v_Normal, lightVector), 0.1);
    // Add attenuation.
    diffuse = diffuse * (1.0 / (1.0 + (0.25 * distance * distance)));

    // Multiply the color by the diffuse illumination level to get final 
    gl_FragColor = v_Color * diffuse;
}
```

With per-fragment lighting, our fragment shader has a lot more work to do. We have basically moved the Lambertian calculation and attenuation to the per-pixel level, which gives us more realistic lighting without needing to add more vertices.

FURTHER EXERCISES

Could we calculate the distance in the vertex shader instead and simply pass that on to the pixel shader using linear interpolation via a *varying*?

WRAPPING UP

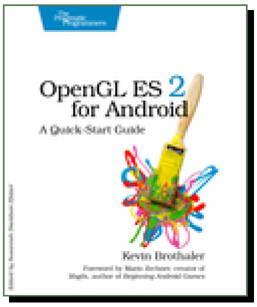
The full source code for this lesson can be downloaded from the project site on GitHub.

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

As always, please don’t hesitate to leave feedbacks or comments, and thanks for stopping by!

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



Buy Now

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 23, 2011 / Android, Android Tutorials / ambient lighting, attenuation, diffuse lighting, fragment shader, Gouraud shading, interpolation errors, Lambertian reflectance, per-fragment lighting, per-pixel lighting, per-vertex lighting, quadratic attenuation, shaders, vertex shader

45 thoughts on "Android Lesson Three: Moving to Per-Fragment Lighting"

pleyas

July 4, 2011 at 8:52 am

Thanks for your help and support!
This series of tutorials are really a good help!

Will be good to have texture and multi texture tutorials as well… are they coming?

Admin

July 8, 2011 at 2:15 am

Hey Pleyas,

Thanks for the comment! They should be the next group of tutorials to come out… in hopefully not too distant of a future. 😉

Admin

September 25, 2011 at 6:09 am

First texture article is finally up! Sorry about the wait. In my absence I realized there's a bit more demand for this than I thought, so I'll try to keep up. 😉

Amplify91

July 21, 2011 at 9:17 am

These tutorials are great! Please continue them! It's very hard to find good information on ES 2.0 for Android. I'm looking forward to the lessons about textures!

Admin

October 2, 2011 at 9:19 pm

First texture article is up! Check out Lesson Four and let me know what you think. Could spin out many articles on textures probably depending on the direction that people would like to see.

Amphiprion

August 16, 2011 at 2:32 pm

Hello,
First of all, many thanks for your tutorial.

Is it possible that you provide a code sample (snippet) using textured cube instead of colored cube.

Thanks in advance.

Admin

October 2, 2011 at 9:18 pm

Lesson Four is up now. 🙂

Spirit

September 5, 2011 at 8:55 am

good lessons i've ever seen!
but for some reason, i cann't download the source code.
could you plz send me a copy~
thx so much~~
zhxjunspirit@gmail.com

Admin

October 2, 2011 at 9:18 pm

Hi Spirit,

I sent you a zip by email. Also, please check the GitHub as I'm with a different repository now.

sinu

September 10, 2011 at 7:02 pm

good tutorials.. had been looking for some and all i found was related to iPhone.. i know its mostly device independent but you need a starting point right! these tutorials really showed how i could start openGL ES 2 in Android 🙂
Please continue the series with texturing, sprites, accessing hardware features etc.. i can help, even though i am not an expert 🙂

Admin

October 2, 2011 at 9:17 pm

There really doesn't seem to be that much info out there for Android. Hopefully this site can help to fill in some of those gaps.

I would definitely appreciate any contributions — the GitHub project is open to forking and the license is Apache, so it's yours to fork and modify as you wish! 🙂

Ryan Chavez

September 21, 2011 at 11:32 pm

Any word on when new tutorials will be made available? I would love to see one about object loading or model animation.

Keep up the good work!

Admin

October 2, 2011 at 9:15 pm

Hi Ryan,

Thanks for the feedback! New tutorials are slowly streaming onto the site. Object loading/model animation is a good one. Probably a few lessons before I get there. 😉

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

Brandon Peters

December 21, 2011 at 8:24 am

Just a heads up regarding the fragment shaders in this tutorial. Multiplying the color by the diffuse value will also multiply the 4th component of the color: the alpha. The non-shader source for this tutorial doesn't have alpha blending enabled (and honestly I'm unsure of OpenGLES/Android's capabilities in this area) so it's mostly a non-issue but it's still good practice to make sure your shaders only influence the values you intend (the color in this case, not the opacity). Just reset the alpha value to 1.0 after the multiply like so:

gl_FragColor = v_Color * diffuse;
gl_FragColor.a = 1.0;

On the other hand, you can get some really sweet effects by making objects see-through based on how much "light" is hitting them.

Admin

March 9, 2012 at 3:20 pm

I'll have to try some of those effects out sometime. Thanks for the comment. 🙂

Ryan Chavez

March 9, 2012 at 6:12 pm

Would it be a better solution to use the "rgb" accessor so that you maintain the current frag color's alpha?

So instead of this:
gl_FragColor = v_Color * diffuse;

gl_FragColor.a = 1.0;

You can use this:
gl_FragColor.rgb = v_Color.rgb * diffuse.rgb;
// alpha is retained
// gl_FragColor.a = 1.0;

Ally

March 1, 2012 at 1:03 pm

I do trust all the concepts you've presented on your post. They are really convincing and will definitely work. Still, the posts are very quick for novices. May you please extend them a bit from next time? Thanks for the post.

Admin

March 9, 2012 at 3:22 pm

Thx for the feedback! 🙂

And Cos

March 9, 2012 at 8:37 am

Hello,
Thank you for your tutorials, they helped me allot.
I havae a question regarding v_Position in fragment shader: in vertex shader we pass the vertex position a_Position and then we transform it to eye space: v_Position = vec3(u_MVMatrix * a_Position).
My question is: in fragment shader, how does v_Position becomes fragment position and not vertex position ?
Thank you.

Admin

March 9, 2012 at 2:43 pm

This is a good question! The key is the way we defined this in the vertex shader:

varying vec3 v_Position;

The "varying" bit means that when this value is passed to the fragment shader, it will be linearly interpolated across the three vertices that make up this triangle.

If we pretend we're in a 2D world, then we might have the following 3 v_Positions:

(0,0)
(1,0)
(1,1)

For a triangle that sort of looks like this:

```
    /|
  /  |
/    |
------
```

The fragment shader will be called for all of the fragments of that triangle, and for each one, the v_Position will be an interpolated value. A fragment around the middle of the triangle might have a v_Position of (0.5, 0.5). There is nothing special about the name "v_Position"; what makes this work is the fact we defined this as a "varying". We could linearly interpolate any vector using the same method.

Let me know if this helps out. 🙂

And Cos

March 9, 2012 at 5:56 pm

Thank you very much, i really appreciate your tutorials and replies,they are very well explained ,in detail and they helped me understand better.

heartdisease

April 11, 2012 at 12:13 pm

Very nice tutorial! Your code lacks something important though – the normal matrix. One does not simply transform the normal vector into eye space 😉
To get the normal matrix you have to invert the model-view matrix and transpose it then. The model-view matrix only applies to vertices. This article tells you why: http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/

Admin

April 12, 2012 at 4:56 pm

Very good point! I believe this is a concern only if you add scaling; if you stick to translations and rotations, then you don't need to invert and transpose the matrix. The bottom of that article seems to suggest as much.

To be fair it's been a long time since I touched this math, so let me know. 🙂

skan

October 7, 2016 at 11:44 pm

I actually need to be able to scale my object and right now it looks like the lighting doesn't work very well with scaling. The object appears to be really flat when scaled down and then gets all those beautiful shadows etc when scaled up. Some questions:
How would I get the normal matrix and at what point would I use this normal matrix? Most information out there is for OpenGL/ES 1, which I can't quite square with the shaders approach. Any links would be appreciated!

Thanks a lot and I love the tutorial!

Steve

April 23, 2012 at 10:43 pm

Yes you are correct. The OpenGL 1.x uses the inverse-transpose modelview matrix to transform normals but this is a very expensive calculation to perform per vertex and is only necessary if you are performing non-uniform scaling. Usually with pre-generated models you are only performing uniform scaling so this calculation can thankfully be avoided. 🙂

Oscar

April 26, 2012 at 11:45 pm

After 2 days I got to this point.
I now need to practice and make my own exercises and examples.

I can't wait to get to the other examples;

I'm curious about how you managed to know what to use on each code blocks of your code?
I mean, did you went further and investigated about how Per Vertex lighting was done before investigating further about and finding info of the lambrt factor?

Also do you think OpenGL would do good with Collision detections?

Since you can pretty much interact with the GPU by the GLSL language I assume you could also use some boolean values to make validations unless there are no "IF" statements on the GLSL.

Can't wait to get into VBOs, I can't figure out what they are in comparissions to FloatBuffers since as far as I know they are Buffer OBjects!

Thank you again for this unique tutorials.

Admin

May 29, 2012 at 6:09 am

Hi Oscar,

Sorry for missing this! To be honest, I didn't really know what to do at first. I just read and read, and looked at both theory and practice from other sites, and then I kept working on it until it worked and looked right. 😉 I think that's one of the best ways to learn IMO — with something like OpenGL, at least for someone like me, it's easier to "get it" by actually coding away until it makes sense to you. Once you've done that, go back and look at the theory, and see if things mesh up.

For collision detection, this would actually be something outside of the purview of OpenGL and lie within your engine. At least that's usually how I see it done. It depends what you want to do though.

You can also use IF statements but they are limited in OpenGL ES 2. Another thing is that they slow down execution, because the branch limits parallel execution. You can definitely use IF statements though — my latest tutorial uses an IF to do different lighting for the front and back face.

Nathan Schagen

May 28, 2012 at 7:50 pm

Hi,
First of all, nice tutorials!
Second, I believe I've spotted a small error. You are using the ModelView matrix to transform normals, while a special normal matrix should be used. I'm talking about this line:

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

Please have a look here:

http://www.lighthouse3d.com/tutorials/glsl-tutorial/the-normal-matrix/

It only gives problems when non-uniform scaling is done with the matrix, so it might not actually be a big problem here. This might be worth mentioning in this tutorial though.

Other than that, everything is really good and easy to understand. 🙂

Nathan Schagen

May 28, 2012 at 7:51 pm

Oops.. I just noticed somebody allready mentioned this. 😉

Admin 👤

May 29, 2012 at 6:03 am

Thanks for letting me know, Nathan! I probably should mention something in the tutorial just to clear things up, since you and the other guy are both right 🙂

sujith

September 4, 2012 at 11:12 am

Thanks for providing these wonderful tutorials. I understood few things better with these tutorials which couldn't understand by reading some reference books. 🙂

Admin 👤

September 4, 2012 at 4:27 pm

Thank you for the compliments 🙂

Pixs

September 6, 2012 at 5:25 pm

Hi!

I was juste wandering if the interpolated normal vector in the fragment shader should be
normalized…
To apply Lambert's Law correctly, we should compute dot product of two units vectors. In your fragment shader, the interpolated vector is 'at best' a unit one.

If normals don't vary that much along triangle vertices, error may be negligeable.
But maybe normalizing normal is worth the effort in case of smoothed low poly meshes ?

Admin

September 6, 2012 at 8:22 pm

Yes, that's a good point. With a wide variation it might also be better to not interpolate surface normals between vertices.

quantumcafe

November 27, 2012 at 3:44 pm

I was wondering, what's the solution to the further excercise? Is it correct, that the lightPos is constant for everey fragment, thus (in the vertex shader) we could calculate the distance by subtracting v_Position after transforming it to an eye space? And the distance, declared as varying, would be interpolated automatically?

Did I get that right? 😛
regards

quantumcafe

November 27, 2012 at 3:49 pm

Or, do we need the interpolated v_position, in which case this wouldnt work..?

Admin

January 30, 2013 at 3:12 pm

It's been a while, but basically anything that has a linear relationship can be safely calculated in the vertex shader and passed via a varying. For example, the distance has a linear relationship because it's based on the position, and since we're working with triangles, the position can always be interpolated linearly between two vertices.

Once you move into non-linear behavior, like exponential falloff, dot products and so forth, then it's better to calculate these in the fragment shader, as linear interpolation of non-linear quantities can lead to artifacts.

BitsMcGee

June 14, 2016 at 9:54 pm

Are you sure that is an accurate statement? I don't think distance is linearly dependent on the interpolated positions, i.e. length(lerp(v1, v2, delta)) != lerp(d_v1, d_v2).

BitsMcGee

June 14, 2016 at 9:55 pm

* length(lerp(v1, v2, delta)) != lerp(length(v1), length(v2), delta).

Chris

January 24, 2013 at 12:52 am

These are excellent tutorials, but where is the specular component!? This isn’t yet complete Phong shading. Other than that, excellent.

Admin

January 30, 2013 at 3:10 pm

Yep, no specular component. That will be in a future lesson, or when I redo this lesson. 😉

FiniteResource

September 18, 2014 at 7:11 am

These are perfectly paced lessons.

karl

May 6, 2015 at 2:52 pm

Hey I can´t get out why the lightpoint is moving / moving as it is. Please can anyone tell me. Probably something really easy, but really bothers me.

Pingback: [Android]OpenGL Study | Dion's Note

Learn OpenGL ES / Proudly powered by WordPress