What is it?

Ambient occlusion is a technique where surfaces are shaded dependant upon how blocked (or ***occluded***) from the surrounding environment they are. The effect is a soft shading around edges, underneath close-proximity geometry and in holes and grooves where ambient light can't reach, much like the effect produced by radiosity and global illumination.

These occluded locations are also likely places where dust and grime would probably collect, hence sometimes this technique is also referred to as ***dirt shading***. Below is an object shaded with an occlusion shader in LightWave, giving at a somewhat dirtier feel then normal.

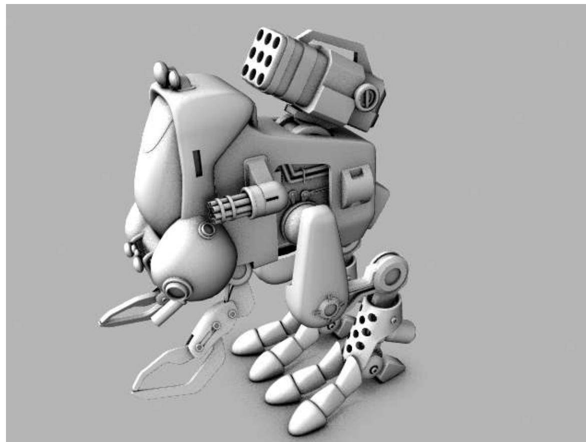

Illustration 1Dirty look by occlusion shading

Hows it work?

The technical aspects behind ambient occlusion are fairly simple, though some ways in which the effect is implemented may differ. This explanation is fairly broad, but hopefully it makes the idea clearer for anyone interested.

As each pixel on a surface is being rendered, a number of rays are shot out in varying directions from the pixel, in a *hemisphere* around the pixel. When the surface is occluded, these rays will hit other surfaces on the object if they lie within this hemispherical area. For example, the rays for a pixel in a groove of an object will most likely hit the edges of the groove many times. A pixel on a flat surface however, will likely never hit anything unless it is close to the edge of outcropping geometry.

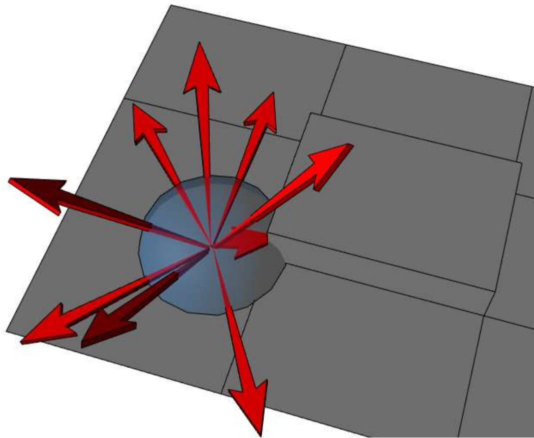

Illustration 2 Rays fire outwards from a pixel

Statistics are recorded of how many rays hit other surfaces and how close these surfaces are, and this information is then used to calculate just how much to shade the surface by.

Making use of it.

So, now we know what it is and how it works (in Laymans terms anyway), how can we make use of it? Ambient occlusion adds those subtle shadows that can only be achieved with the more time-consuming radiosity rendering, but obviously without the render hit (in most cases) that radiosity causes.

Shadow and shading Subtleties

Ambient Occlusion is ideal for adding shading subtleties into renders without the need for radiosity calculations. Shaders can be used to enhance the existing surfacing of a 3D object, or it can be rendered out as a separate pass (or sequence of images) and then applied back into the 3D render with a compositing package.

The latter approach is usually more economical and flexible since a compositing package will allow for quick tweaks, different blend effects and more without the need for any re-rendering of the 3D scene.

Subtle shading and shadows created by Ambient occlusion can add a more realistic feel to 3D rendered items, which makes this technique a great idea for enhancing the look of renders quickly and easily.

Surface Shading

Ambient occlusion information can be used to enhance and create more realistic surface effects when rendering. As we mentioned, ambient occlusion shades in areas where ambient light can't reach.

This would also mean that its possible other things cannot see these areas

and if it can't be seen, gunk and grime probably would collect here as well... Gunk and grime not only can discolor a surface, but obviously other parameters such as reflection, specularity and the like will be affected.

Using occlusion as some kind of control map for surface parameters is definitely a good way to use this technology.

Special illumination effects

Perhaps a less considered use of occlusion mapping is to generate illumination effects – By shading the area under an object, then using the occluded pixels as a control map for luminosity, its possible to quickly fake the effect of illumination from an object.

A shadow that is instead colourised luminous green suddenly becomes a self-illumination effect!

Doing it in LightWave...

LightWave has a few techniques available for Ambient Occlusion. This includes the rendering engines' own radiosity calculations, as well as a few shader plugins created by talented users online that also allow for occlusion effects. This section will briefly review each technique.

Radiosity

The most obvious method is to use LightWave's global illumination capablities to perform an Ambient Occlusion pass. This is done fairly easily by removing all local illumination in the scene (ie deactivating all lights), setting the backdrop color as white and activating Radiosity with the 'Backdrop only' option in the global illumination panel.

Dependant upon the model, the values can quickly be baked into either vertices, or an image using LightWave's SurfaceBaker plugin. This is a common technique used to show off nice models with that now-familiar 'clay' look.

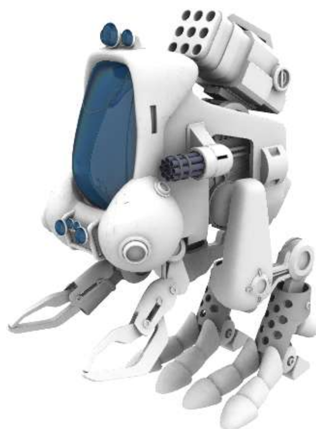

Illustration 3Radiosity shows off the modelling detail

Plugins : gMIL

gMIL, by developer Eric A Soulvie, is a plugin solution for performing ambient occlusion. While it has shipped with LightWave for some time now, there was not much information about its use and essentially it meant that only the more adventurous users tended to use it.

However, setting up ambient occlusion is relatively easy, and being a plugin it means that you do not need radiosity, and the effect can be blended nicely into existing surfaces. However, because it is a surface shader, it must be added to all surfaces that you wish to have ambient occlusion. For complex surfaced models with many surfaces, this may be quite time-consuming to set up.

Using the plugin:

Select the surface that you wish to add the gMIL shader to. Add the shader, then open its properties panel to set up options.

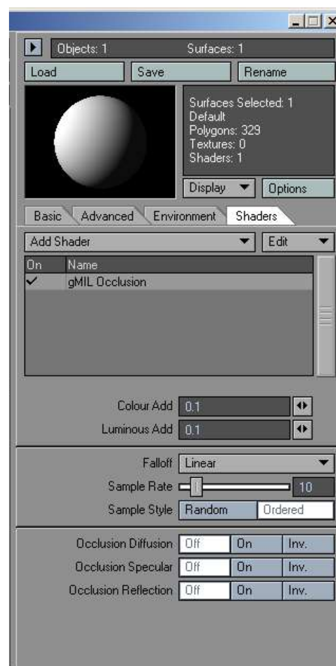

Illustration 4gMIL's properties

To get going, simply activate the *Occlusion Diffusion* option by clicking *On*. This will apply the occlusion effect to the diffuse channel. Clicking *Inv* will invert the effect, making the surface black and the shading bright.

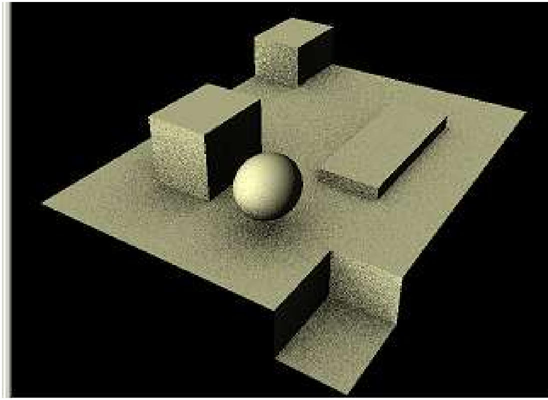

Illustration 5Default gMIL diffuse

To tighten up the effect, change the *Falloff* to *y=x^2*. This will change the sampling from being so broadly spread across the surface to being more intense closer to the pixel. If the effect is too grainy, then upping the *Sample Rate* will improve the quality, but at the expense of increasing render times.

You can change the *Sample Style* to *Ordered*, however this gives an effect that appears to look like many, many raytraced shadows that have gone a little haywire, and will require a high sample rate to get good results from.

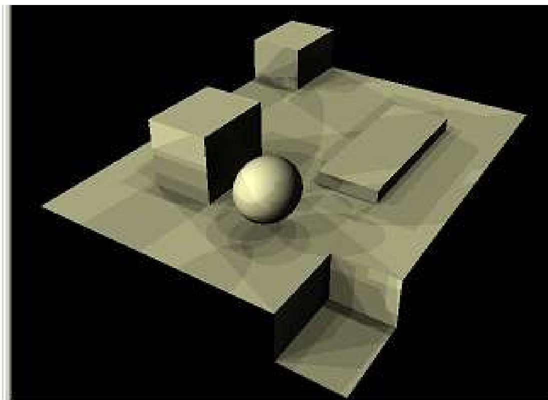

Illustration 6Sample Style Ordered

Plugin : SG_AmbOcc

Just recently released, this plugin is fantastic and produces a very clean result. It can be downloaded freely from http://www.informatik.hu-berlin.de/~goetsch/AmbOcc/ (or can be found on the plugin database at http://www.flay.com)

Using the plugin:

Select the surface that you wish to add the SG_AmbOcc shader to. Add the shader, then open its properties panel to set up options.

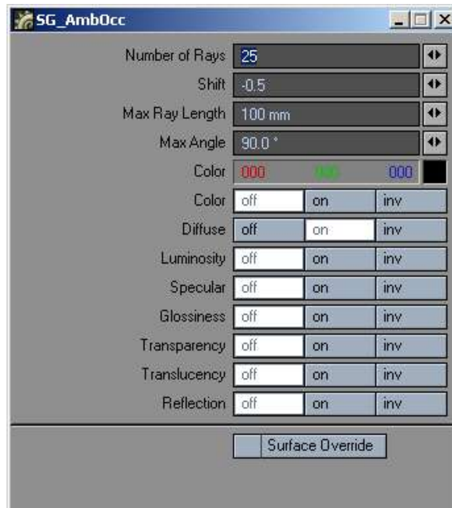

Illustration 7SG_AmbOcc's properties

By default, the plugin is set up to affect the diffuse channel. Simply press F9 to test the settings to see how the effect works.

If you don't see much effect, only in grooves and edges, then its likely that the value for *Max Ray Length* needs to be increased. This value specifies the range for the rays to test for. Short values are good for creating occlusion only within fine edges and grooves on a model, a trait fround in dirt and grime.

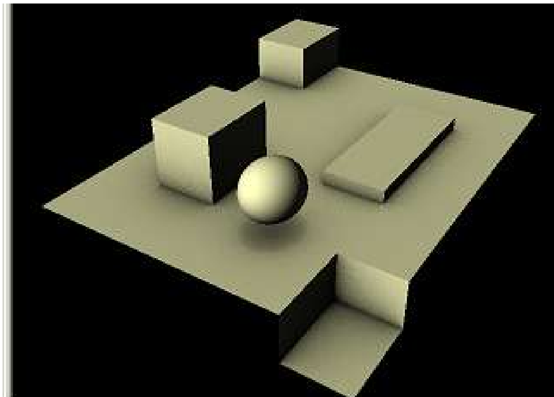
Illustration 8Default Settings used

If you want to ignore the edges and grooves on a surface, and only create occlusion from geometry above the surface, then lower the *Max Angle* parameter. By default, rays are always fired out in the direction of the pixels normal (this is 0 degrees). A *Max angle* of 90 allows rays to be fired out horizontally and vertically flat along the surface. By lowering this value, you force the rays to be directed upwards and ignore low edges and outcropping geometry.

If the effect is a little too grainy for your liking, then adjust the *Number of Rays* parameter accordingly. Note that like gMIL, setting this value too high will increase rendering times.

The *Shift* value controls the intensity of the effect. To lighten the darkening

effect of the diffuse channel for example, simply lower this number (ie. From -0.5 to -0.1)

Note that you can also apply occlusion to most surface attributes and these can be activated through the *On* or *Inv* options next to their name. The color attribute uses the *Color* values, while the others use the *Shift* values.

As with gMIL, Inv inverts the occlusion effect.

The last parameter, Surface Override, is a special option to white-out a surface and apply occlusion shading to the surface. This is a great way to render flat white images with just occlusion present (as it ignores diffuse shading, etc), and makes doing so very quick and easy without the need to modify surface .

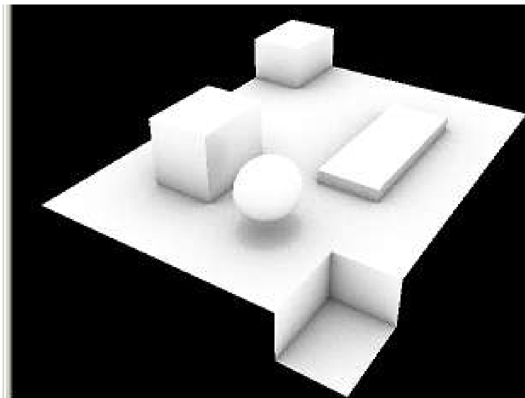

Illustration 9Surface Override

Static Headaches be gone – A Tip!

Because a lot of sampled rays in all the solutions mentioned use random values, there can be a bit of noise in rendered animations. While upping the number of rays used to sample can greatly lower the graininess, it does mean that rendering times can increase. If you're on a timeframe to deliver a job, this may not be desirable.

Here is a small tip you can use to get cleaner results with Ambient occlusion shaders and radiosity and not end up with unreasonable render times.

- Set the number of rays in the shader, or radiosity, to get a reasonable result with fast render times. Don't go overboard attempting to get a perfect look, just a fairly reasonable quality.

- Set the camera to an antialias level of low enhanced.

- Deactivate 'Adaptive Sampling' completely. Adaptive Sampling will just force LightWave to antialias the first rendered pass. Without this active, each AA pass will be recaclulated... And as random values are being used, this means each AA pass will be different then the last.

- Render. If the quality is not quite up to scratch, either raise the number of AA passes, or increase the rays.

Without Adaptive Sampling active, each AA pass will be different, and when all AA passes have been rendered, LightWave then blends them together into the final image. The graininess of each pass will be softened substantially to give a cleaner result with low rays.

This trick works well but doesn't always guarantee results, so you should experiment with this trick and decide for yourself if its producing the result you are after.

Writing an Occlusion Shader in Lscript

The following code demonstrates some ideas and approaches to developing an Ambient occlusion-style shader in Lscript, LightWaves own scripting language. This script is very basic, it doesn't always produce a great quality result, but it does demonstrate the principles behind writing an occlusion shader.

There are two methods for scanning rays around each pixel, and two methods for shading the surface. Each method has various pros and cons. Play with them to see the differences (they are selectable in the GUI)

The example code below is fairly well commented, and hopefully shouldn't cause too many headaches when trying to read and understand it.

```
//----------------------------------------------------------------------------
// LScript Occlusion Shader
// (C)opyright 2004 Kevin Phillips
//
// Performs a simple ambient occlusion calculation and adjusts
// the diffuse channel.  This code is provided as-is for learning from, and
// there are no guarantees on code quality or performance.  You use this
// code at your own risk...
//
// 21-Sep-04    KP  * Enhanced code, with a couple of scanning and shading
//                    approaches...  Added GUI as well...
//----------------------------------------------------------------------------

@version 2.2
@warnings
@script shader

// global values go here
numAdjust = -1.0;       // Max value that can be applied to surface
numRayLength = 0.5;     // Max distance to scan for
intRndRays = 8;         // Number of rays to shoot out (Random version)
intScnRays = 32;        // Number of rays to shoot out (Scan version)

flgScanType = 0;        // Flag for scan type (0 = random, 1 = scanner)
flgCalcType = 0;        // Flag for calculation type (0 = distance, 1 = hits)

currentTime,currentFrame;

create
{
    // one-time initialization takes place here
}

destroy
{
    // take care of final clean-up activities here
}

newtime: frame, time
{
    // called each time the current time index changes in the scene

    currentFrame = frame;
    currentTime = time;
}

init
{
    // called at the beginning of each render pass
}

cleanup
{
    // called at the end of each render pass
}

flags
{
    // let's Layout know which attributes of the surface you will alter
    // In this case, its the Diffuse channel (and Raytrace cause thats what we're doing!)

    return(DIFFUSE,RAYTRACE);
}

process: sa
```

```
{
    // Grab the object ID.
    objectID = sa.objID;

    //---------------------------------------------------------------------------
    // Ignore the preview sphere to prevent the shader from crashing.  The Preview
    // sphere in the surface editor has no normals, and LScript tends to crash when
    // trying to render it...
    //---------------------------------------------------------------------------
    if (objectID.id != 268468223)
    {
        // Grab the shaded pixel position in world coordinates, and the normal direction.
        saPosition = <sa.wPos[1], sa.wPos[2], sa.wPos[3]>;
        saNormal = <sa.gNorm[1], sa.gNorm[2], sa.gNorm[3]>;

        // Grab the current diffuse value of the point
        saDiffuse = sa.diffuse;

        //---------------------------------------------------------------------------
        // To check for Ambient Occlusion, simply shoot out rays in a hemispherical
        // shape from the pixel being shaded, calculate if rays hit any other surface
        // and use this information to shade the pixel.
        //
        // I have implemented a couple of experimental approaches here for you to
        // try...
        //
        // RANDOM - Fires rays randomly in various directions.
        // 4 x 8 SCAN - Fires 8 rays around the hemisphere, tilting 0, 22.5, 45, 90 degrees

        hits = 0;   // Reset the hits counter
        dist = 0.0; // Reset the ray distance counter

        if (flgScanType == 0)
        {
            //---------------------------------------------------------------------------
            // RANDOM VECTOR APPROACH
            // This can look quite noisy!
            for (i = 1; i <= intRndRays; i++)
            {
                // Get a random direction
                fireRay = RandomVector(saNormal);

                // Shoot the ray out to see what it hits...
                shootRay = raycast(saPosition,fireRay);

                // Did it hit something?
                if (shootRay != -1.0 && shootRay <= numRayLength)
                {
                    hits++;             // Inc the hit counter
                    dist += shootRay;   // Add the distance of the ray to statistics
                }
            }
        } else
        {
            //---------------------------------------------------------------------------
            // 4 x 8 SCAN VECTOR APPROACH
            // Scan 8 around the hemisphere x 4 upwards, can be slow...
            for (y = 0; y <= 1; y += 0.25)
                {
                for (z = -1; z <= 1; z += 0.5)
                    {
                    for (x = -1; x <= 1; x += 0.5)
                       {
                            // Get a direction
                            fireRay = ScanVector(saNormal,<x,y,z>);

                            // Shoot the ray out to see what it hits...
                            shootRay = raycast(saPosition,fireRay);

                            // Did it hit something?
                            if (shootRay != -1.0 && shootRay <= numRayLength)
                            {
                                hits++;             // Inc the hit counter
                                dist += shootRay;   // Add the distance of the ray to statistics
                            }
                        }
                    }
                }
        }


        // Calculate the shading value after finishing the scan.
        numPercentage = 0;

        //---------------------------------------------------------------------------
        // I've included two methods for calculating the shading in here (like offering
        // two methods for scanning...
        //
        // DISTANCE - Takes the distance from the pixel to another hit point, and shades
```

```
        // as a percentage based on falloff...  Can be noisy with random approach.
        //
        // HIT - A random approach that shades based on how many hits the rays made.
        // This approach can look 'gluggy' with the 4 x 8 scan...

        if (hits > 0)
        {
            if (flgCalcType == 0)
            {
                //-------------------------------------------------------------------------
                // DISTANCE AVERAGE
                // Calculate via distance. This is calculated by the average distance of the
                // rays that hit.  This length is then worked out as a percentage of the max
                // ray, and then inverted. Inverting the percentage makes the shorter rays
                // more intense (and hence darker)

                avgDist = dist / hits;         // Calculate the average distance of all the hits
                thePercentage = avgDist / numRayLength; // Work out the percentage...
                numPercentage = 1.0 - thePercentage;    // Invert the percentage

            } else
            {
                //-------------------------------------------------------------------------
                // HIT PERCENTAGE
                // Calculate via hit percentage.  This is calculated by the number of hits
                // that were made to other surfaces, then multiplying the amount to apply to
                // the surface by this value. With scan mode, this can be very messy...

                if (flgScanType == 0)
                {   // Random vector statistics
                    numPercentage = hits / intRndRays;
                } else
                {   // Scan vector statistics
                    numPercentage = hits / intScnRays;
                }
            }
        }


        //-------------------------------------------------------------------------
        // FINISHED : We simply take our percentage and work out how much 'adjusting' to
        // do to the surfaces own diffuse value.

        // Create the amount of diffuse adjustment
        adjDiffuse = numAdjust * numPercentage;

        // Shade the diffuse value at this point.
        sa.diffuse = (saDiffuse + adjDiffuse);
    }
}


//-------------------------------------------------------------------------
// NOTE : No saving or loading options are added yet because of the experimental
// nature of this code... Feel free to add them in if you wish! :-)
load: what,io
{
    if(what == SCENEMODE)   // processing an ASCII scene file
    {
    }
}

save: what,io
{
    if(what == SCENEMODE)
    {
    }
}

options
{
    reqbegin("Test Occlusion Shader (C) Kevman3d.com");

    c1 = ctlpopup("Scan Type",flgScanType + 1,@"Random Rays","8 x 4 Search"@);
    c2 = ctlpopup("Shade type",flgCalcType + 1,@"Distance falloff","Hit percentage"@);
    c3 = ctlnumber("Maximum diffuse adjustment",numAdjust);
    c4 = ctlinteger("Rays to scan (Random mode only)",intRndRays);
    c5 = ctlnumber("Max ray distance to scan (m)",numRayLength);

    return if !reqpost();

    // get requester control values here
    flgScanType = getvalue(c1) - 1;     // Value is 0 or 1
    flgCalcType = getvalue(c2) - 1;     // Value is 0 or 1
    numAdjust = getvalue(c3);
    intRndRays = getvalue(c4);
    numRayLength = getvalue(c5);

    reqend();
```

```
}


// -------------------------------------------------------
// UDF to calculate a random vector
// -------------------------------------------------------

RandomVector : worldNormal
{
    // Create a random directional vector
    VX = randu() - randu();     // -90 to 90 degrees
    VY = randu();               // 0 to 90 degrees
    VZ = randu() - randu();     // -90 to 90 degrees

    // Convert to actual normal direction for LightWave to use
    nX = (VX * worldNormal.y) + (VY * worldNormal.x) + (VZ * worldNormal.z);
    nY = (VX * worldNormal.z) + (VY * worldNormal.y) + (VZ * worldNormal.x);
    nZ = (VX * worldNormal.x) + (VY * worldNormal.z) + (VZ * worldNormal.y);

    finalVector = <nX, nY, nZ>;

    return(finalVector);
}


// -------------------------------------------------------
// UDF to calculate directional vector
// -------------------------------------------------------

ScanVector : worldNormal, directionVector
{
    VX = directionVector.x;
    VY = directionVector.y;
    VZ = directionVector.z;

    // Convert to actual normal direction for LightWave to use
    nX = (VX * worldNormal.y) + (VY * worldNormal.x) + (VZ * worldNormal.z);
    nY = (VX * worldNormal.z) + (VY * worldNormal.y) + (VZ * worldNormal.x);
    nZ = (VX * worldNormal.x) + (VY * worldNormal.z) + (VZ * worldNormal.y);

    finalVector = <nX, nY, nZ>;

    return(finalVector);
}
```