

## Baking Next-Gen Lighting with Turtle

In this tutorial we are going to check out the light baking tools that are available in Turtle 5.1.




**ILLUMINATE LABS™**

LIGHTING THE NEXT GENERATION OF GAMES


The Illuminate Labs Team has been working hard on this new release in order to get you the most advanced lighting techniques available at your fingertips.

We'll check out all the different stuff crammed into Turtle, like Radiosity Normal Maps, Polynomial Texture Maps and a lot more.

The tutorial is divided into sections, so you don't have to do it all at once. If you already know the basics you should be able to skip ahead to the more fun bits.

 Load the provided scene [hangar.mb](#). This is a pretty detailed scene that we can use to

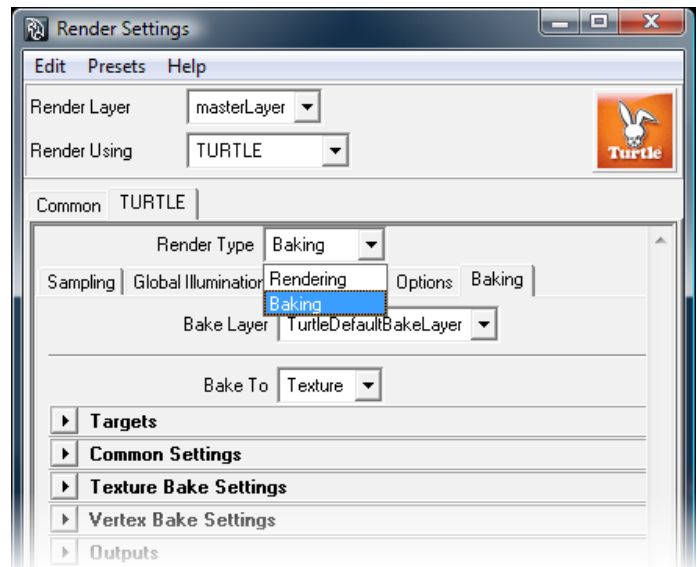
test out the baking tools on.

 If you render out a frame, you should get a basic render of the hangar building. This will be the base for the first baking tests.



## Baking Tools in Turtle

If you haven't checked out Turtle's baking tools yet, you should know that they are an extensive part of the Turtle toolset. You find the main baking tools, together with the Turtle Renderer, as a separate tab inside the Maya Render Settings when you enable Turtle as the active renderer. There is also a Point Cloud Bake Editor located under [Window → Rendering Editors → TURTLE](#). The baking toolset is divided into three categories,




- **Texture Bake** – render effects to a surface texture and transfers details from one surface to another
- **Vertex Bake** – render out lighting and other data to the vertices of a surface
- **Point Cloud Bake** – transfer lighting and other data to an arbitrary set of points

The huge range of different baking tools and settings aren't there just to confuse you, they all have their specific uses, and they actually complement each other, since specific baking features are only available in specific modes. We will touch on most of the baking tools in some way in this tutorial, but we won't cover all the features in one go. No need in overloading the gray cells too much, right? Check the Turtle Reference Manual for all the greasy details on the different baking tools.

The base of the hangar has been prepared for baking, check out the Display Layers in Maya to see the different components. The [hiResBase](#) layer contains all the detailed geometry we'll be baking down to the rough geometry contained in the [lowResBase](#) layer. The other parts of the hangar, like the roof and the vents, are contained in the [hiResScene](#) layer.

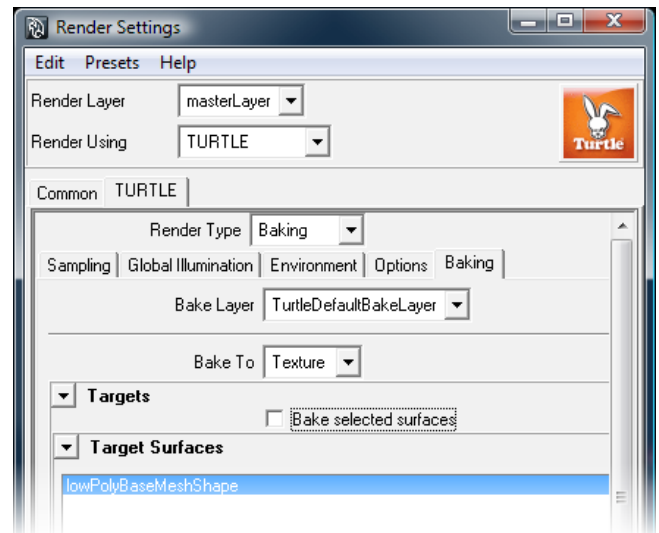
## Surface Transfer

We'll start out by baking a normal map, which we'll make use of later. The [Baking](#) tab holds settings specifically used for baking but all the different features that were enabled for standard rendering will also be used for all the baking tools.


 Open the [Global Illumination](#) (GI) tab and disable global illumination, otherwise we may get a GI pre-pass when we are baking, even though we only want to output normal maps. Now it's time to check out the Surface Transfer functionality, so switch the [Render Type](#) from [Rendering](#) to [Baking](#) and open the [Baking](#) tab.

Turtle baking tasks are organized into different Bake Layers that can quickly be changed between. You create and organize your baking layers in the [Turtle Bake Layer Editor](#) found under [Window → Rendering Editors → TURTLE](#). We'll only be baking the base of the hangar, so the default layer will do.


Doing Surface Transfer is a fairly straightforward process. Imagine that you get a messy Maya scene, with millions of polys or gigantic shading networks. And imagine that you want to sort this out in some way so you can make use of the content for your game or some other application.



All you have to do is sketch out some low-poly proxy geometry that roughly approximates the scene, UV-map it, and you're good to go. The low-res surface goes into the [Target Surfaces](#) list, all the detailed nastiness goes into the [Source Surfaces](#) list. You won't even have to touch UV-maps or other stuff on the high-res geometry, because Turtle will sample the scene from the UV-map specified by your low-res surface!


 Open the [Display Layer Editor](#) in Maya and make sure the [lowResBase](#) layer is visible. All [Target Surfaces](#) and [Source Surfaces](#) need to be set to visible for the Surface Transfer to succeed. Right-click the [lowResBase](#) layer and click [Select Objects](#). In the [Target Surfaces](#) roll-out, choose [Add Selected](#) to prep the low-res geometry for baking. Each [Target Surface](#) maintains its own list of [Source Surfaces](#), so make sure the low-res geometry object is selected in the list. Add the objects in the [hiResBase](#) layer to the [Source Surfaces](#) list by right-clicking the layer in the [Display Layer Editor](#), picking [Select Objects](#), then choosing [Add Selected](#) in the [Source Surfaces](#) roll-out.

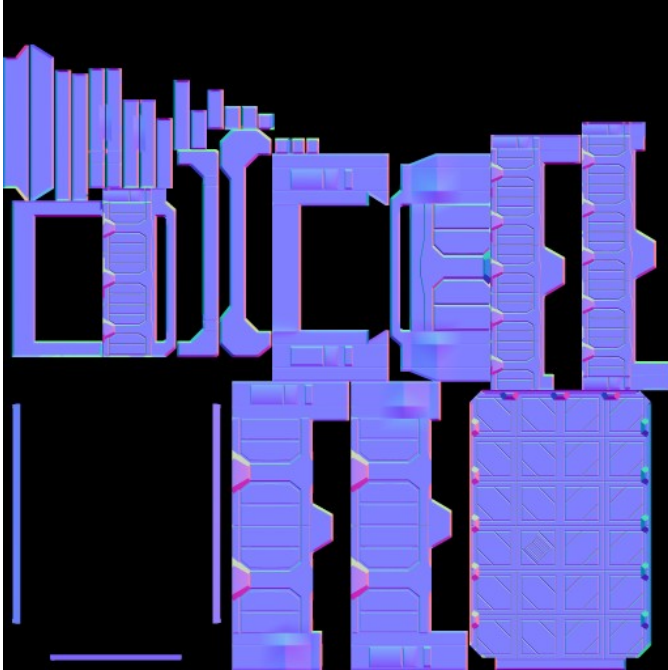
The [Outputs](#) should already be set by default to render out only [Normals](#), which is what we want in this case. We have to tweak the [Transfer Settings](#), because we have to tune the Surface Transfer to pick up the [Source Surfaces](#) that are the farthest away from the [Target Surface](#). [Front](#) and [Back](#) options controls if sampling is done in the direction of the normals of the Target Surface, or in the opposite direction. We actually want to start sampling a slight distance above the Target Surface, and sample down towards the surface, so we will only use back sampling.

 Set both [Front Range](#) and [Front Bias](#) to 0.0, which will stop sampling in the normal direction. Set the [Back Bias](#) to -2.0, which will start sampling from a distance of 2.0 above the surface in the normal direction. Set the [Back Range](#) to 3.0; this will make Turtle sample 2.0 units towards the surface, and 1.0 units beyond the surface, so that we pick up any penetrating geometry.

Don't worry too much about the Bias and Range parameters. Just remember that the Bias parameters offset the search ray origin, the Range parameters control the max distance to search for Source Surfaces, and Front and Back respectively represent sampling away from and towards the surface in respect to the normal.

It is often a good trick to choose either Front or Back Sampling, and set the Range attribute to double the Bias attribute, which will sample the same distance from the surface on either side, but with rays going in a uniform direction.

 You should now have **Front Range: 0.0**, **Back Range: 3.0**, **Front Bias: 0.0** and **Back Bias: -2.0**. In the **Texture Bake Settings** roll-out, increase **Width** and **Height** to 2048, and hit render.




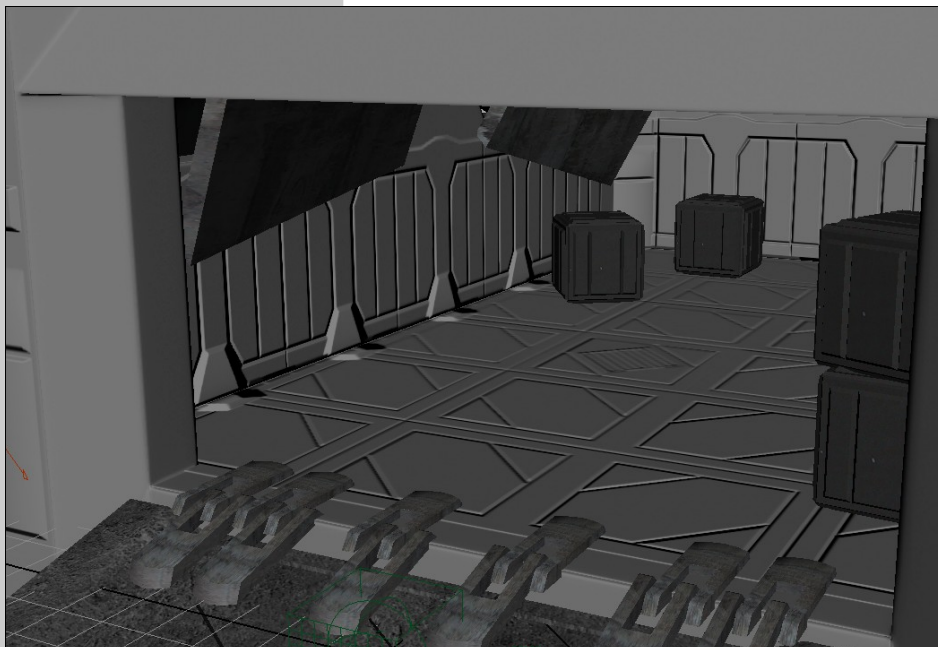
The baked normal map contains the geometrical details from the high-resolution **Source Surfaces**, stored as image data in an ordinary RGB image.

The super sampling of the normal map can be controlled in the **Anti-Aliasing** roll-out in the **Sampling** tab of the **Turtle Render Settings**. You can also select regions in the **Render View** to re-render, if some areas need higher sampling.


Take a peek at the **Outputs** roll-out in the **Baking** tab. You can bake down almost anything with Turtle. You can bake down any custom shader and even control Turtle through LUA scripting, so baking normal maps is just the start of what Turtle can do!

You can visualize the normal map inside Maya by connecting an **ilrHwBakeVisualizer** shader, but it's often easier to check **Model View Hardware Visualization** in the **Texture Bake Settings** roll-out of the **Baking** tab. A hardware shader will be connected to your object after rendering. Any software shader on the object will remain untouched.


 Check **Model View Hardware Visualization**, and render out a new texture. Remember to enable **Hardware Texturing** in the **Shading** menu of your Maya viewport.




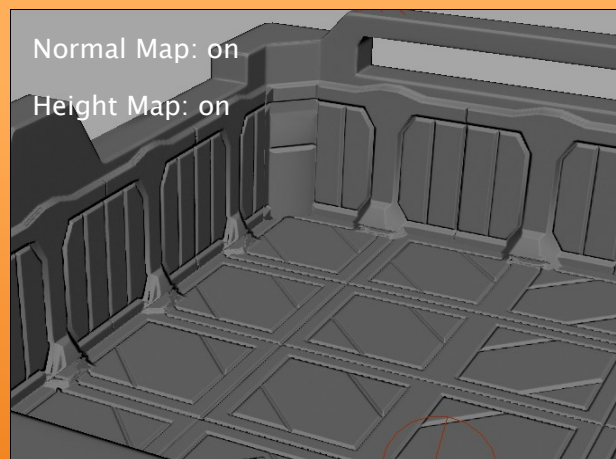
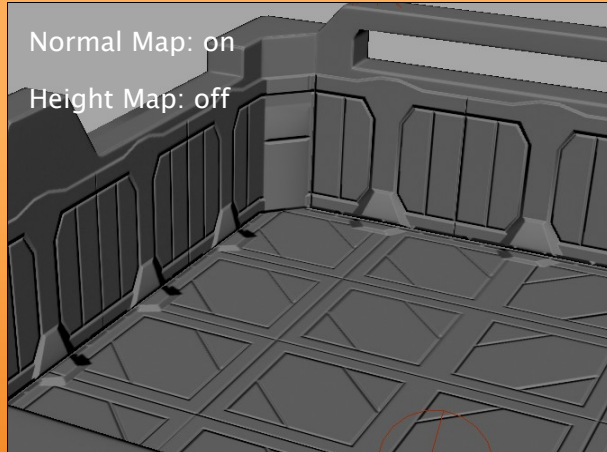


 The bump maps on the high-res geometry do not contribute to the normal map by default, so check **Include Bump Maps** in the **Outputs** and bake out a normal map with bump maps as well. Rename the old map before you render and store away both maps so we can make use of them later.

## Parallax Mapping

 We can check out one more really simple trick before we proceed with Turtle's new baking features. Uncheck the **Include Bump Maps** attribute to get the simple flat shading in the normal maps. Check the **Displacement in Alpha Channel** instead, and Turtle will now render out a black-and-white height map into the alpha channel of the normal map. If you still have the **Model View Hardware Visualization** checked, Turtle will automatically connect the map for you and switch on the **Height Map** in the hardware shader, so go ahead and hit render. Remember that both **Source Surfaces** and **Target Surfaces** need to be visible for a Surface Transfer!

 The default values for the height map should be a bit too high for this scene, so dig your way down to the hardware shader for the object and set the **Scale** in the **Height Map** roll-out to 0.04. Since the displacement values are fit into the range [0, 1] by default after rendering, and we get negative displacement values in this scene, you will also need to set the offset value to 0.33. Go ahead and check the base mesh from different angles. Notice how there is a better illusion of depth now? The pixel-shader on the object actually ray-traces the object locally, which creates an approximate displacement effect on the surface.





A useful trick for making the effect of parallax mapping slightly more pronounced is lighting the surface from different directions with slightly different colors. Also notice the shading artifacts along the true edges of the low-res mesh. As the effect is only local to the surface, the details which seem to be displaced out from the surface will still be cut-off at the edge of the mesh. You should probably model those protruding blocks on the walls with geometry. It's a good thing to keep an eye out for any problem areas and try to plan ahead. If you use parallax mapping wisely, you can make your work really shine.


## Radiosity Normal Mapping

We're going to bake out a **Radiosity Normal Map**, a technique that was extensively used in Half-Life 2. In a **Radiosity Normal Map** (RNM), lighting (direct and indirect!) is baked down for every point as one RGB color for each of three different directions. This makes it possible to interpolate the specific lighting by weighting among the three color samples appropriately, depending on the normal of the surface. This will make it possible to swap normal maps on our hangar base, and still use the same RNM for lighting look-up, producing high detail lighting.

All you need to bake down an RNM is the low-res geometry receiving lighting. As you might recall, Texture Bake just samples down any property you want into a texture map which you can then reapply on the source surface of the baking.


 Open the **Display Layer Editor**, and hide the **hiResBase** layer; we'll not need it again since we have already baked our normal maps. Create a new point light with **Intensity** 2.0, and linear **Decay Rate**. Make sure the light uses ray traced shadows. Set the color to bright green, and position the light at the back of the hangar towards one of the corners. Neat Area 51 lighting, huh?

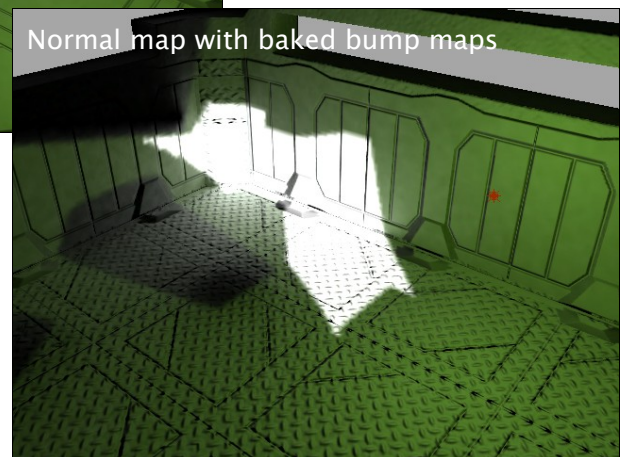
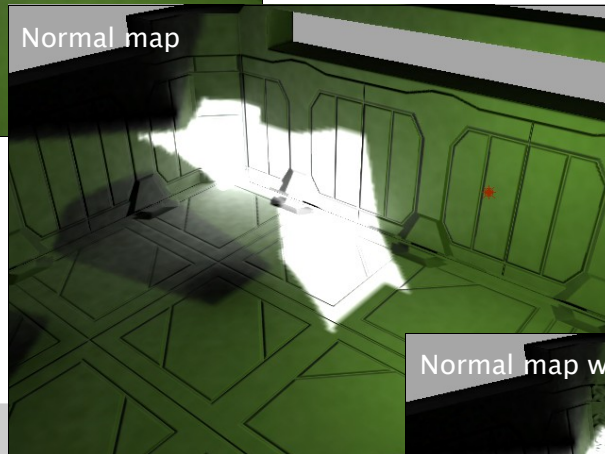
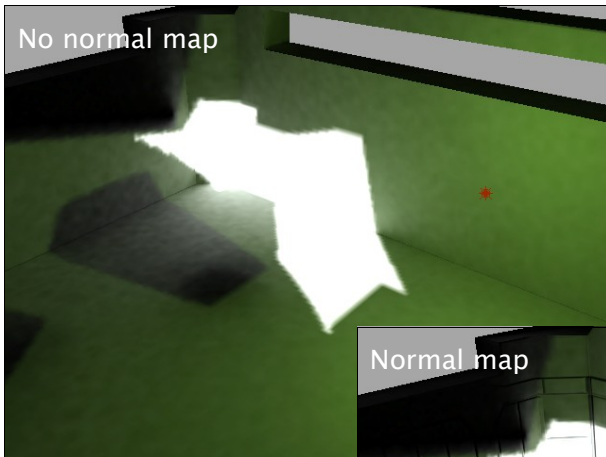
 The default Bake Layer is OK, since the low-res geometry should already have been added to it when we baked the normal maps. Open up the **Outputs** roll-out and make sure only **Radiosity Normal Map** is checked. Everything should now be prepped for an RNM texture bake, so go ahead and hit render.

 We will use the scene from this point later on, so go ahead and save the scene. Use another name if you don't want to overwrite the original scene file.



You will only see one image in the [Render View](#), but if you check the output directory you will actually have three files, each one representing lighting for one of the RNM basis vectors. Note that the images are somewhat noisy. To improve the quality you can increase the [Samples](#) setting for RNM baking. A better way is to use the [Radiance Cache](#), which we will cover in the next section.


 The [ilrHwBakeVisualizer](#) shader has built-in support for RNM maps, so you can either plug them in there, or simply check [Model View Hardware Visualization](#) in the [Output File](#) roll-out of the [Texture Bake](#) tab before you render.




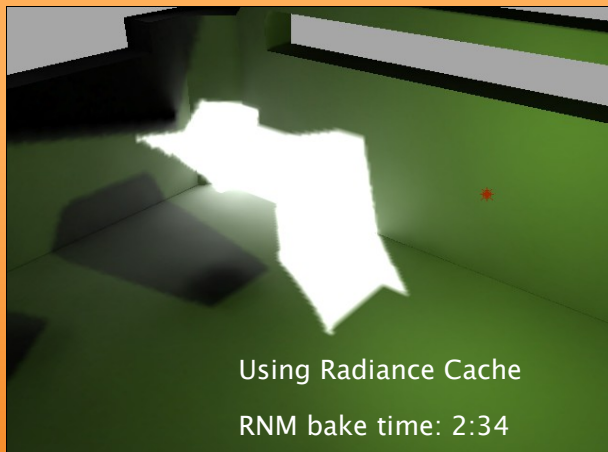
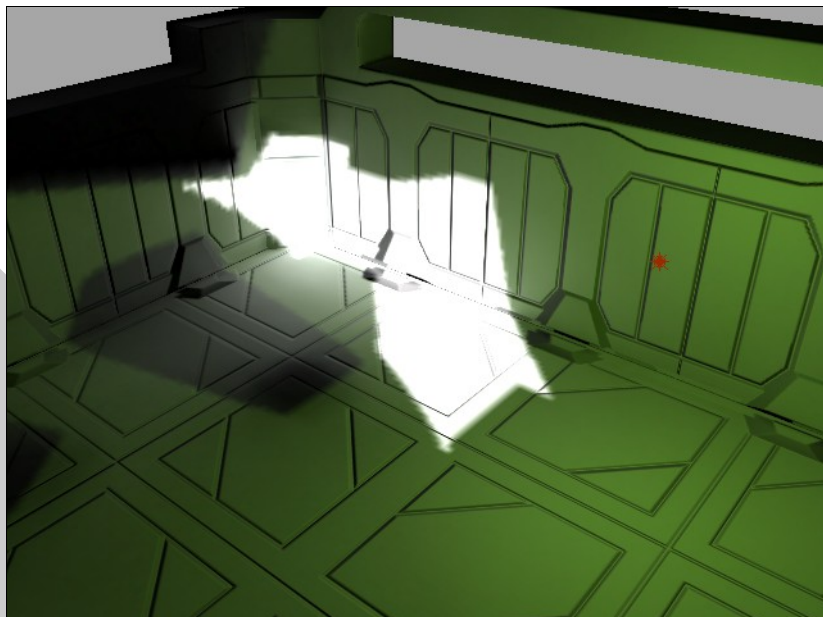
If you enabled visualization when we baked the normal maps, you might already have a normal map visible, otherwise you can locate the visualizer shader in the [Hardware Shader](#) roll-out of the active material. Remember to enable [Normal Map](#) in the Hardware Shader Features if it's not already enabled. Go ahead and try the normal maps we baked earlier together with the RNM. Since the RNM contains all the light of the scene, we need to turn off all lights (Use [No Lights](#) in the [Lighting](#) menu of your Maya viewport) and hide the [hiResScene](#) Display Layer. In the images above, the material color has also been set to a brighter color than default in the hardware shader. No matter which normal map you apply, the lighting will update to match. Such is the magic of RNMs! Even though the RNM might be very low in actual pixel dimensions, it will still adapt to the normal map and shade accordingly, which is a trick used extensively in Unreal Engine 3 and next-gen games.

## Radiance Cache

Let's see how we can produce RNMs with higher quality at faster rendering speeds using Final Gather in combination with a Radiance SH cache.

 In Turtle's **Render Settings**, open the **Global Illumination** tab and enable GI. Choose **Final Gather** as **Primary GI** and set **Secondary GI** to **None**. This will give the RNM one bounce of indirect light. For more information on how to use secondary GI for deeper indirect light, see our tutorials on GI. Open the **Final Gather** roll-out and set **Use Cache** to **Radiance SH**. Change the number of **Gathering Rays** to 200.

 Go ahead and hit render. Turtle will now present a prepass before the main render, which in turn will produce a nice and smooth RNM.



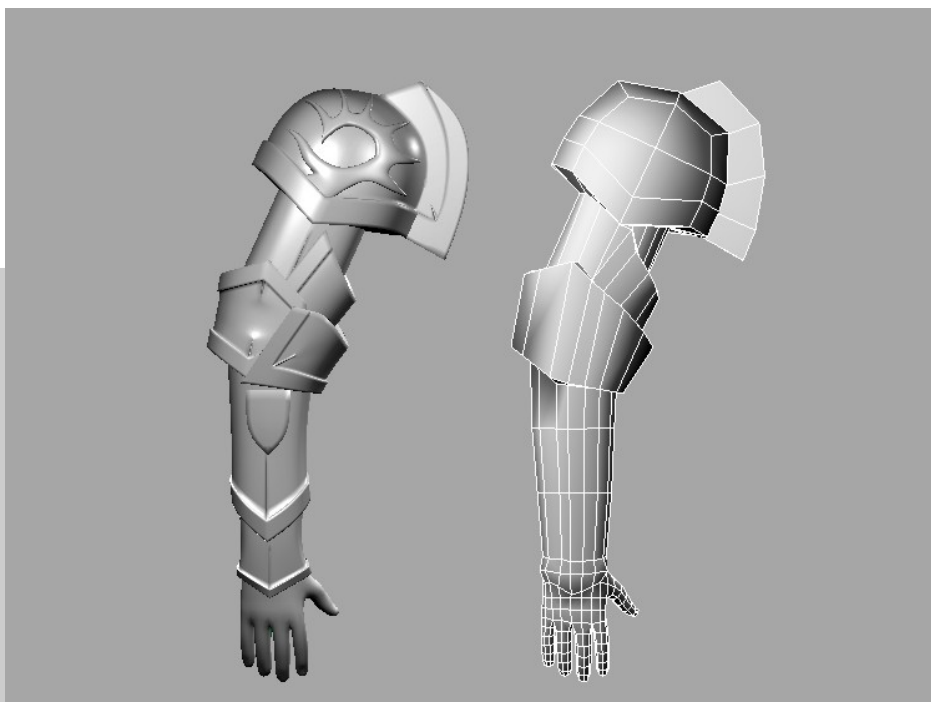



## Polynomial Texture Mapping


You might already be familiar with PTMs, but we'll go ahead and take a peek at how it works in Turtle anyway. The PTM works by baking down diffuse illumination to a simple polynomial function basis, which is written out as two RGB image maps. The PTM will contain an approximation of the diffuse shading from any given light direction, including shadows and self-shadowing. PTM is available as a separate roll-out from the [Baking](#) tab's output section.


You can bake down simpler illumination and shadows using regular texture baking. This works well if, for instance, you have a classic sphere on a plane situation, and want to bake down the shadows from the ball onto the plane. If we want to capture self-shadowing, then [Surface Transfer](#) is the choice, since in most cases you want to sample the subtle self-shadowing of a detailed object, and bake it down for use on a low-res substitute.

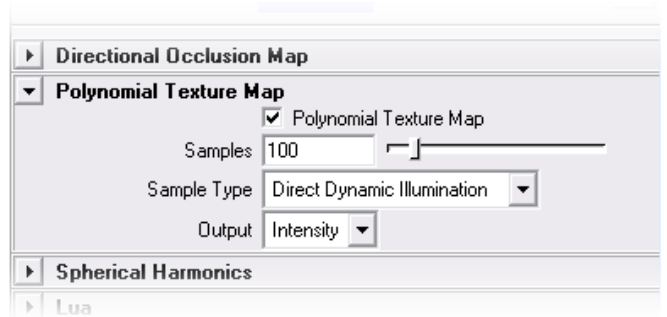
 We're going to bake out some PTMs for a medium-detail model of an arm. Go ahead and open the provided scene [armor.mb](#).




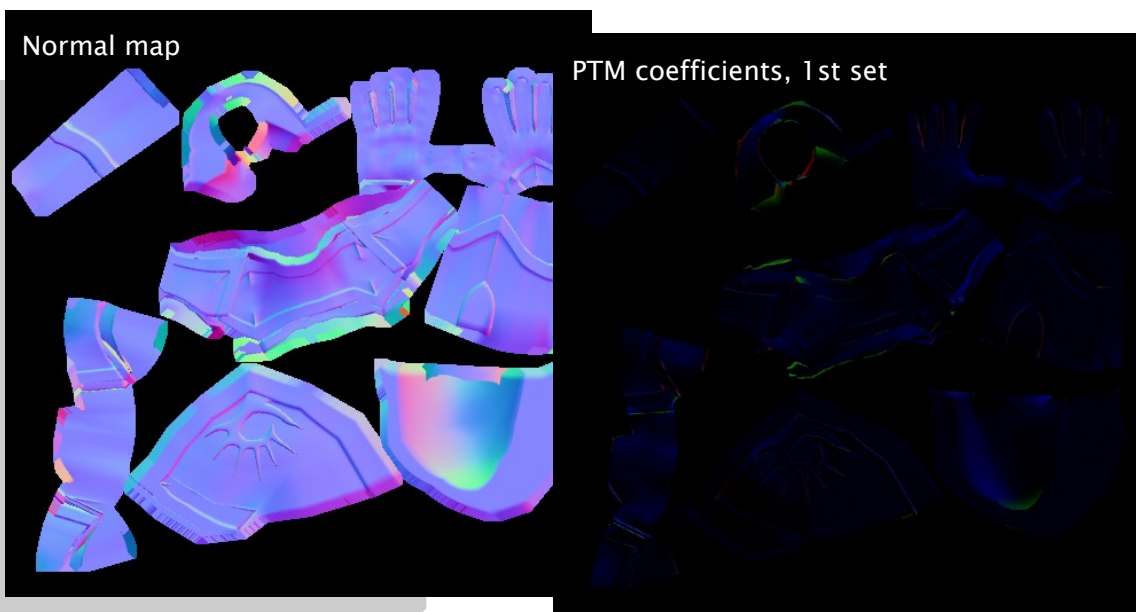
 The scene is divided into two Display Layers, the [highres](#) and the [lowres](#) layer contain the Source Surfaces and the Target Surface respectively. Make sure both layers are visible, and switch the [Render Type](#) to [Baking](#) in Turtle's render settings. Head over to the [Baking](#) tab and set the low-res geometry as [Target Surface](#). Add all the high-res surfaces to the [Source Surfaces](#) list. As we have multiple overlapping objects in the high-res model, you should sample the surfaces towards the interior, so let's set [Front Range](#): 0.0 and [Front Bias](#): 0.0 to stop all outward rays. [Back Range](#): 4.0 and [Back Bias](#): -1.0 will make sampling begin at a slight distance from the surface of the armor, and keep sampling a little past the low-res surface, to make sure we don't miss any Source Surfaces in difficult areas.


 In the **Texture Bake Settings** roll-out, increase the resolution to 1024x1024, and *always* remember to select a floating-point image format when baking PTM; **OpenEXR MultiLayer** will be fine. The PTM coefficients can have any possible value, so storing them as ordinary images will require a reparameterization of some form, and a resulting decrease in precision.


 Check **Model View Hardware Visualization** too, which will save you some time when connecting the PTM components. In the **Outputs** roll-out, enable **Normals** and **Polynomial Texture Maps**. Also make sure that you have Maya's OpenEXR plugin loaded.



 100 PTM Samples will be ok for these tests, but for a final baking you could go as high as 500 or more if you want. You can sample down both direct and indirect lighting in a PTM, but only diffuse effects. View-dependent effects like specular shading will not be baked down. We'll look at baking indirect dynamic lighting later, so the default settings of the PTM roll-out should be ok. One word of caution however; if you're using **Super Sampling**, you should remember to always uncheck the **Clamp Values** attribute (among the super sampling settings), since PTM coefficients can be any possible float value. If super sampling isn't enabled, go ahead and select it. Did you uncheck **Clamp Values**? Fire up a render, and Turtle should bake out the normal map and the PTM.



 Once the maps are baked, you can hide the **highres** layer. Make sure you have **Hardware Texturing** enabled in the viewport, and select **Use All Lights** in the **Lighting** menu of the viewport. You should have a directional light next to the arm, which you can rotate to change the lighting on the arm. To see the effect of the normal map, you should open up the low-res mesh material, make your way to the **Hardware Shader**, and switch on **Specular Color** in the **Hardware Shader Features**.

 There is also an option to [Extract Normals](#), this will recreate the surface normals from the PTM, but since we have already baked the normals into a normal map, lets uncheck this option.

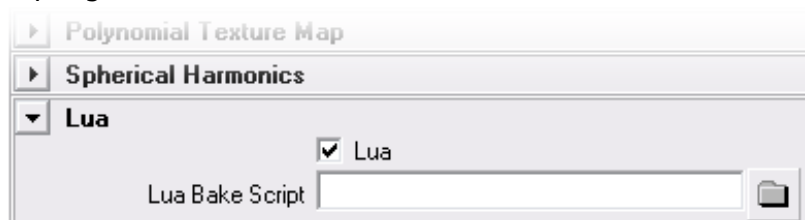


Try rotating the directional light next to the arm. You should be able to spot subtle self-shadowing along the edges of the details.

## Lua Baking

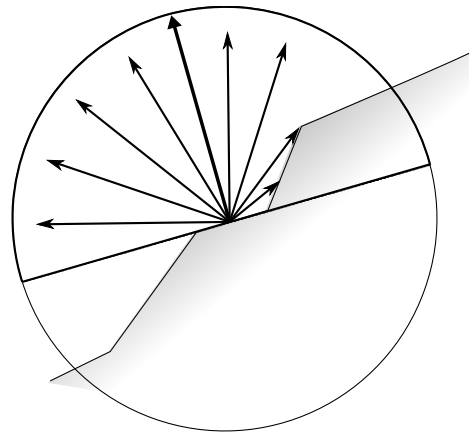
As of version 4, Turtle has been completely restructured to be able to bake all the new lighting passes. If the preset lighting passes doesn't cut it, the same interface that the preset passes are built through is exposed for you so that you can customize it through Lua scripting.


You can customize the behavior of both [Texture](#) and [Vertex](#) baking; the Lua options are located under the [Outputs](#) roll-out.



The Lua bake option is probably not for your everyday work, you'll need patience and some time spent with the Turtle Reference Manual if you want to demystify all the details of the baking interface. But if you consider yourself a Turtle techie, you will absolutely not be able to resist baking out your own custom lighting passes using Lua scripts!

The Lua baking operates in the sphere surrounding any given surface sample. The baking interface lets you tell Turtle exactly what kind of sampling you wish to perform from the sampling point, what relative coordinate system you wish to work in, what kind of basis you wish to project the sampling to, and stuff like that.



 Let's look at how you would bake an ordinary PTM texture with Lua, so you'll get an overview of the parts comprising a Lua Bake Script. We're going to use the [armor.mb](#) scene again, so prep Source and Target Surfaces for a Surface Transfer if they are not already. Switch everything but the Lua baking off in the Outputs. All Lua Baking Scripts must implement one or more of the following functions

- **setup** – configure Turtle, run once before the baking task
- **basis** – define a custom basis, will be evaluated by Turtle during rendering
- **bake** – process samples and other data, run once for every surface sample

Let's take a look at a simple Lua Bake Script.

```
function setup()
    bset("gather.sampletype", "direct dynamic illumination")
    bset("gather.space", "tangent space")
    bset("gather.distribution", "equiareal")
    bset("gather.minsamples", 100)
    bset("gather.maxsamples", 200)
    bset("gather.coneangle", 180)
    bset("gather.cosweighdynamic", true)
    bset("output.size", 6)
    bset("basis.type", "custom")
    bset("basis.rgb", false)
end

function basis(x, y, z)
    values = {x*x, y*y, x*y, x, y, 1}
    return values
end

function bake()
    ci = getBasisCoefficients(0)
    return ci
end
```


The function `setup()` contains a series of `bset` statements, which set the options for the Turtle baking. The Turtle Reference Manual will be your best friend here, since you'll probably not be able to remember all the options from memory. The `gather.sampletype` attribute controls the type of illumination that Turtle will sample; it's currently set to sample direct dynamic illumination. Dynamic illumination means that we wish to consider any possible lighting direction, which is suitable if we wish to bake down a PTM, for example. We can also bake down indirect dynamic lighting, but this requires us to correctly setup a Dynamic Photon Map in the [Render](#) tab, which will generate an emission of photons down onto your scene from all directions.



You can also specify indirect illumination, which will bake down the static indirect illumination in the scene, which is what you'd want for baking an RNM. The `gather.space` attribute defines the coordinate system in which you want Turtle to sample in. It's currently set to the tangent space defined by the low-res mesh, but it can also be set to object space or a coordinate system based on the sampled source surface. The `gather.distribution` attribute controls how the samples are spread out over the sphere, the equiareal option samples equally in all directions, but you can choose to use a cosine weighted distribution as well, which can be useful if you are baking occlusion maps. Turtle will automatically make an adaptive sampling between the `gather.minsamples` and `gather.maxsamples`, so be sure to set both in your Lua scripts, otherwise the default values will kick in, which might give you unexpected results. The `gather.coneangle` attribute will restrict the sampling to a subset of the sphere, so setting it to 180 will result in hemisphere sampling in the chosen coordinate system. You can weigh the actual sample data by a cosine weight by enabling the `gather.cosweighdynamic` property. You have to manually set the `output.size` property for your baking script, because in many cases Turtle will not know how many texture components you want to write out. You select a basis for your sampled information through the `basis.type` property. We've specified that we want to use a custom basis, but there are also predefined basis for PTMs and Spherical Harmonics. If you are only interested in the intensity of the illumination, `basis.rgb` should be set to false, otherwise you could have triple the amount of output components to deal with.

The `basis(x, y, z)` function is really very simple. Turtle will call your basis function with a given vector  $v = [x, y, z]$  that defines a point on the sampling sphere, asking you what values this vector will produce in your custom basis. All you have to do is plug the values into your function and return the number of coefficients set by the `output.size` property. The sampled data will automatically be fitted to your basis during rendering, so there's really nothing much to it. Recognize the basis by the way? It's the standard PTM basis!

The `bake()` function is where most of your custom work is done, although it is very basic in this script. We get the coefficients for the intensity by using the `getBasisCoefficients()` function, and pass them right out for Turtle to write out. Passing a value of 1–3 to the `getBasisCoefficients(...)` function gets you the respective color channel coefficients. For such a simple case as this, we don't actually need to implement the `bake()` function, so it could have been skipped completely.

 With Model View Hardware Visualization enabled, Lua baked textures are automatically connected to the Color property of the Hardware Shader. Go ahead and switch this off and render out a frame using the PTM Lua Bake Script. You will have to manually connect EXR channels 0–2 to Light ABC, and EXR channels 3–5 to Light DEF of the Hardware Shader. With Hardware Shading on, you should now get something you recognize.



Just having duplicated PTM baking through a Lua Bake Script isn't much fun, but we can do some interesting things now that we have the base script going. There's a nice trick in the original [PTM whitepaper](#) called Diffuse Gain that is quite easy to implement. It essentially boosts the effect of the PTM by scaling the PTM coefficients appropriately. Change the `bake()` function to

```
function bake()
    ci = getBasisCoefficients(0)

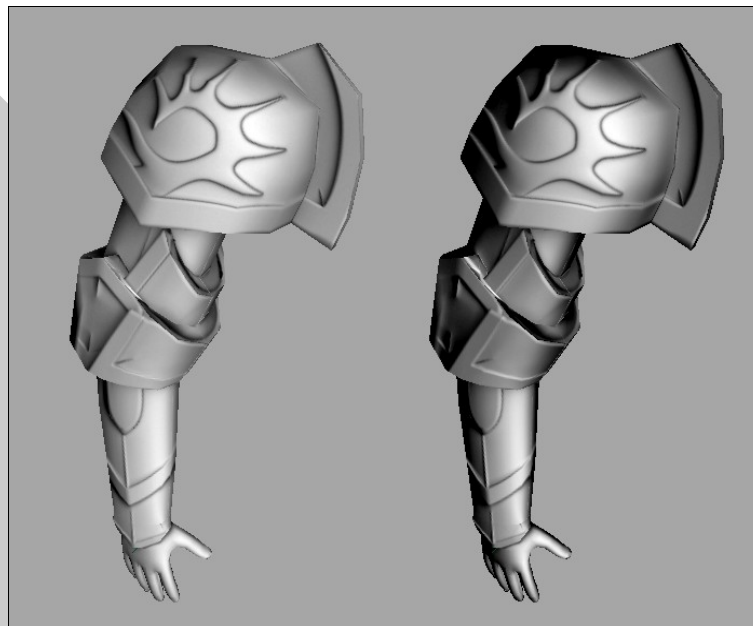
    fMaxX = (ci[3]*ci[5] - 2.0*ci[2]*ci[4]) / (4.0*ci[1]*ci[2] -
        ci[3]*ci[3])
    fMaxY = (ci[3]*ci[5] - 2.0*ci[1]*ci[5]) / (4.0*ci[1]*ci[2] -
        ci[3]*ci[3])

    fLen = fMaxX*fMaxX + fMaxY*fMaxY
    if (fLen > 1.0) then
        fLen = math.sqrt(fLen)
        fMaxX = fMaxX / fLen
        fMaxY = fMaxY / fLen
    end

    diffuseGain = 1.5
    co = {}
    co[1] = diffuseGain * ci[1]
    co[2] = diffuseGain * ci[2]
    co[3] = diffuseGain * ci[3]
    co[4] = (1.0 - diffuseGain) * (2.0*ci[1]*fMaxX + ci[3]*fMaxY) + ci[4]
    co[5] = (1.0 - diffuseGain) * (2.0*ci[2]*fMaxY + ci[3]*fMaxX) + ci[5]
    co[6] = (1.0 - diffuseGain) * (ci[1]*fMaxX*fMaxX + ci[2]*fMaxY*fMaxY +
        ci[3]*fMaxX*fMaxY) + (ci[4] - co[4])*fMaxX + (ci[5] -
        co[5])*fMaxY + ci[6]


    return co
end
```

The `diffuseGain` property controls the amount of boosting; a value of 1 will give standard PTM, unboosted, so try out some different values. Render out a frame and compare to the old PTM without boost to see the effect.



Taking full advantage of Turtle using Lua Bake Scripts will require you to spend some time on the theory and details of newer computer graphics concepts such as Spherical Harmonics, Wavelets or other applicable mathematical tools. For your everyday needs you might be better off using the preset baking passes, but you never know, the need to develop your own custom illumination model might arise sooner than you think! The Lua Bake Scripts will allow you to experiment way faster than when having to write your own proprietary tools.

## More Lua Baking

 Let's look at one final example of how to bake down indirect lighting with the Lua Baking Scripts. We're going to check out the school-book example of baking an RNM using a custom Lua script. Return to the saved scene we worked through previously (modified [hangar.mb](#)). Make sure that **Render Type** is set to **Baking** and head over to the **Baking** tab.

Check out the RNM baking script

```
function setup()

    bset("gather.sampletype", "indirect illumination")
    bset("gather.space", "tangent space")
    bset("gather.distribution", "equiareal")
    bset("gather.minsamples", 100)
    bset("gather.maxsamples", 100)
    bset("gather.coneangle", 180)
    bset("output.size", 12)

end

function bake()

    -- RNM basis vectors
    basis1 = vec3(0.8165, 0.0, 0.577)
    basis2 = vec3(-0.408, 0.707, 0.577)
    basis3 = vec3(-0.408, -0.707, 0.577)

    -- Project indirect light onto our RNM basis vectors
    rgb1 = getSampleProjectedSum(basis1, false)
    rgb2 = getSampleProjectedSum(basis2, false)
    rgb3 = getSampleProjectedSum(basis3, false)

    -- Weigh the indirect contribution
    n = getSampleCount()
    rgb1 = (rgb1 * 2.0) / n
    rgb2 = (rgb2 * 2.0) / n
    rgb3 = (rgb3 * 2.0) / n

    -- Add direct light contribution
    nlights = getLights()
    for i = 1, nlights do

        col = getLightCol(i)

        if (getLightAmb(i)) then
            rgb1 = rgb1 + col
            rgb2 = rgb2 + col
            rgb3 = rgb3 + col
        else
            localLightDir =
                normalize(worldToGather(getLightDir(i)))


            weight = dot3(basis1, localLightDir)
            if (weight > 0) then
                rgb1 = rgb1 + col * weight
            end
            weight = dot3(basis2, localLightDir)
            if (weight > 0) then
                rgb2 = rgb2 + col * weight
            end
            weight = dot3(basis3, localLightDir)
            if (weight > 0) then
                rgb3 = rgb3 + col * weight
            end
        end
    end

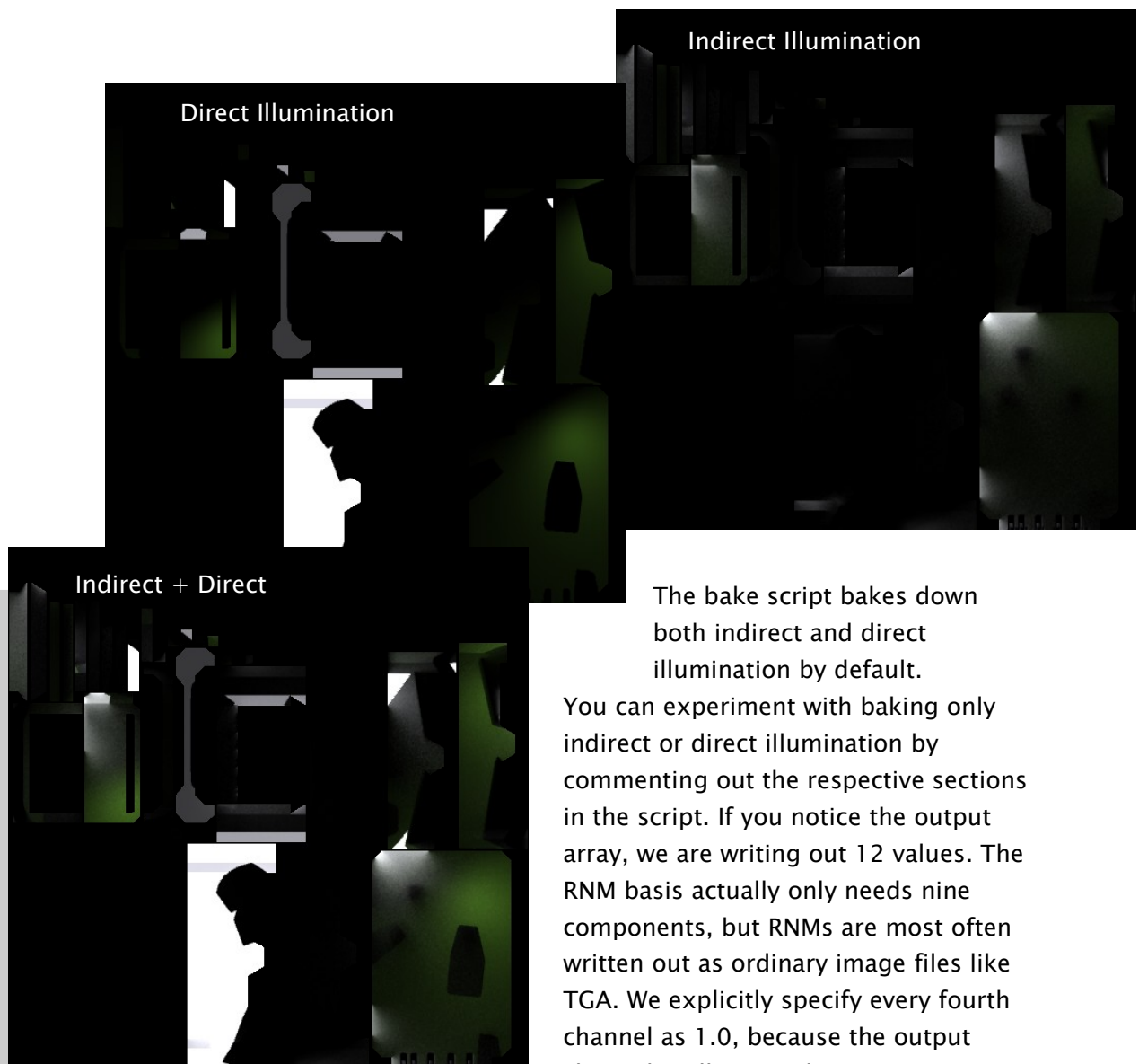
    array = {rgb1[1], rgb1[2], rgb1[3], 1.0,
             rgb2[1], rgb2[2], rgb2[3], 1.0,
             rgb3[1], rgb3[2], rgb3[3], 1.0}

    return array

end
```

The largest difference from PTM Lua baking, which we covered earlier, is that we haven't provided a basis function for this Lua Bake Script. We aren't fitting the sampled data to a function basis, but a simple vector basis, and we can do that ourselves in the bake function. We also choose to sample static indirect illumination instead of dynamic lighting, which will capture bounced light, color bleeding and neat things like that.

 Make sure no other pass is enabled in the Texture Baking tab, enable the Lua pass and bake out a texture using the RNM Lua Bake Script.



The bake script bakes down both indirect and direct illumination by default.

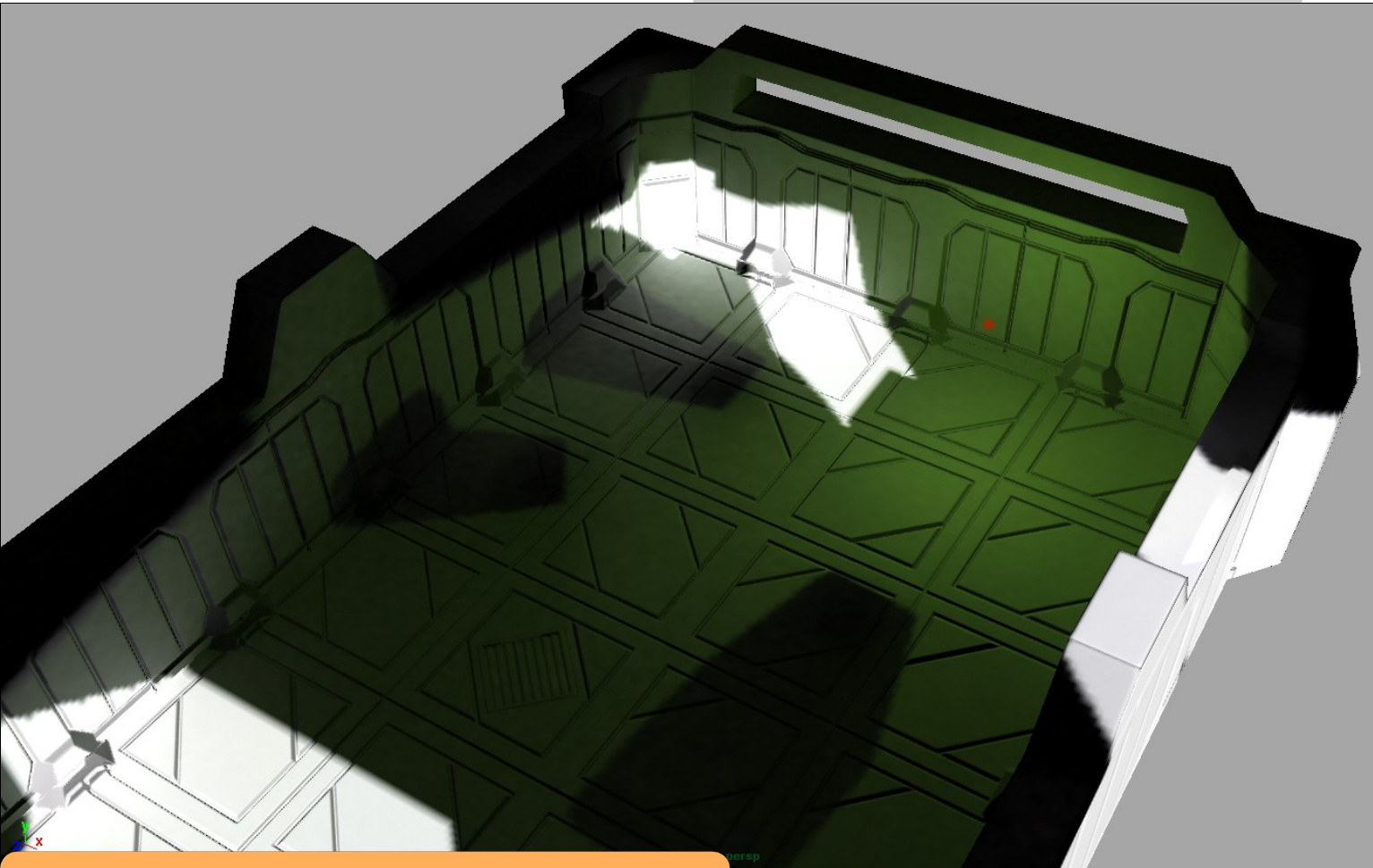
You can experiment with baking only indirect or direct illumination by commenting out the respective sections in the script. If you notice the output array, we are writing out 12 values. The RNM basis actually only needs nine components, but RNMs are most often written out as ordinary image files like TGA. We explicitly specify every fourth channel as 1.0, because the output channels will in turn be written out to

R, G, B and A channels of a set of image files to match the number of output components, so we make sure that all alpha channels will represent an opaque value. Our 12 output components will thus be output to 3 RGBA images, which is what we'd want for an RNM map.

You will really start to appreciate the RNM maps once you start using them on textured scenes, and properly take advantage of color bleeding to pronounce mood or visual style.



Trying out new ideas will be a breeze once you get used to the Lua interface, so get busy coding!



- Normal Mapping
- Parallax Mapping
- RNM – Full Illumination through custom Lua Bake Script

Get your eval license of Turtle now!  
[www.illuminatelabs.com](http://www.illuminatelabs.com)