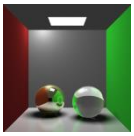


2018/2019, 4th quarter

INFOGR: Graphics / *NeedToKnow*

Lecture 10 – Shaders

Author: Jacco Bikker

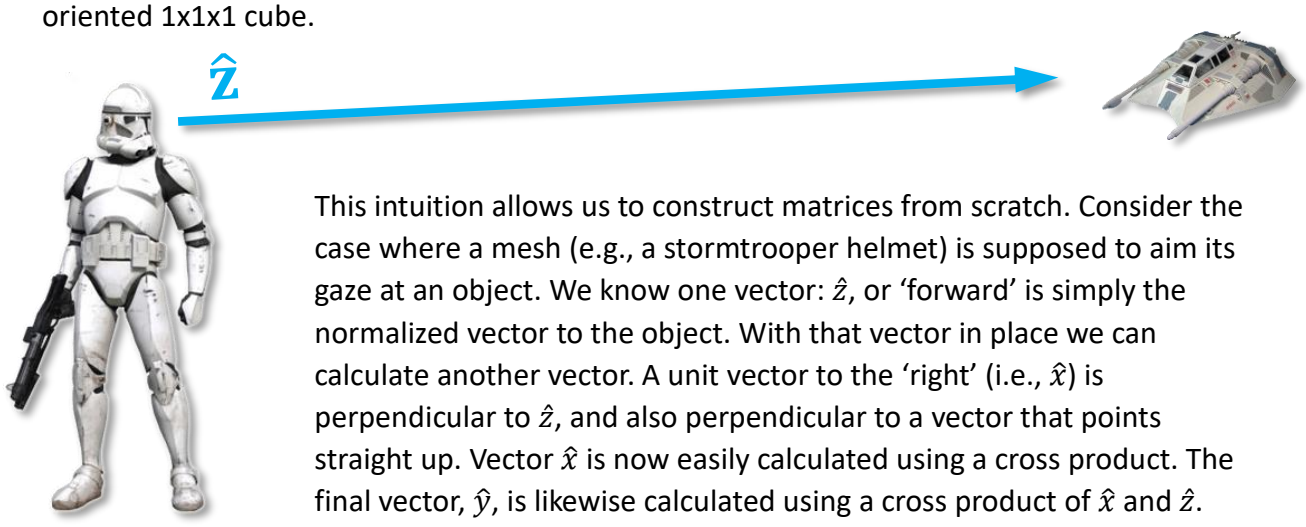
TL;DR

In this installment of 'NeedToKnow': shading. After a brief recap of the basics (from ray tracing), we investigate spaces and the Phong model. We fake reflections using an environment map. We fake details using a normal map, and even fur is faked, this time with an alpha translucency trick. The rasterization world is full of tricks.

Matrices

An important ingredient for the techniques described in this installment of NeedToKnow is the concept of *spaces*. A space, in graphics, is a coordinate system. We encounter them in several places: the screen has a 2D coordinate system, 3D objects that we download from the internet have their coordinates specified in something we call *object space*, and a moving camera brings its own space. When the need arises to have a common space, this is typically *world space*: one space to rule them all. One thing is certain: we will frequently be converting from one space to another. For this, we use matrices. Lots of 'm.

A 3x3 matrix can be seen as a coordinate system. It holds three vectors, which are typically of unit length and perpendicular. In this case, the matrix is *orthonormal*; it's essentially an oriented 1x1x1 cube.



This intuition allows us to construct matrices from scratch. Consider the case where a mesh (e.g., a stormtrooper helmet) is supposed to aim its gaze at an object. We know one vector: $\hat{z}$, or 'forward' is simply the normalized vector to the object. With that vector in place we can calculate another vector. A unit vector to the 'right' (i.e., $\hat{x}$) is perpendicular to $\hat{z}$, and also perpendicular to a vector that points straight up. Vector $\hat{x}$ is now easily calculated using a cross product. The final vector, $\hat{y}$, is likewise calculated using a cross product of $\hat{x}$ and $\hat{z}$.

In proper math:

$$\hat{z} = normalize\left(P_{object} - P_{helmet}\right)$$

$$\hat{x} = normalize\left(\hat{z} \times (0,1,0)\right)$$

$$\hat{y} = \hat{x} \times \hat{z}$$

Matrices allow us to get from one space to another. Positions expressed in a coordinate system can be seen as the sum of distances along the axii of that coordinate system. E.g., $x = y = z = 0.5$ takes us to the center of the unit cube, no matter how it is oriented, by simply summing $\frac{1}{2}\hat{x}$, $\frac{1}{2}\hat{y}$ and $\frac{1}{2}\hat{z}$. The vectors $\hat{x}, \hat{y}$ and $\hat{z}$ obviously are defined with respect to a 'parent' coordinate system, and therefore the matrix now allows us to translate coordinates defined relative to the oriented cube to coordinates in the parent space.

Shading

The concept of spaces allows us to solve an important problem with shaders:

Lights are typically positioned in world space. Vertices are typically positioned in object space, and so are (vertex) normals. We need to transform vertices to orient the object, but we also need to transform the normals. Problem is: to what space should they go? Vertices ultimately end up in camera space, because the camera itself is stationary (because the monitor is stationary). So, in order to perform a simple shading calculation like $N \cdot L$, we either define our lights in camera space, which is inconvenient*, or we transform N to world space.

This means that from now on we propagate *two* matrices while traversing the scene graph. One takes us from object space to world space; the other takes us from object space to camera space. The first one is the concatenation of the transforms of an object and its ancestors; the other is the concatenation of the same object transforms, plus the inverse camera matrix, plus a perspective matrix.

This sounds pretty hard to implement, but in practice, it is quite simple. Consider the following pseudo-code:

```
SceneGraphNode::Render( mat4 toWorld, mat4 toCamera )
{
   toWorld *= localTransform, toCamera *= localTransform // concatenate
   for each child c do c.Render( toWorld, toCamera )
   localMesh.RenderTriangles( toWorld, toCamera )
}
```

We call this code with an identity matrix for 'toWorld', which causes objects without parents to be transformed by their local transform only. We call the code with the inverse camera matrix, combined with a perspective matrix, for 'toCamera'.

---

*: Actually, it is impossible when camera space includes perspective, since our shading calculations must operate in an orthonormal space.

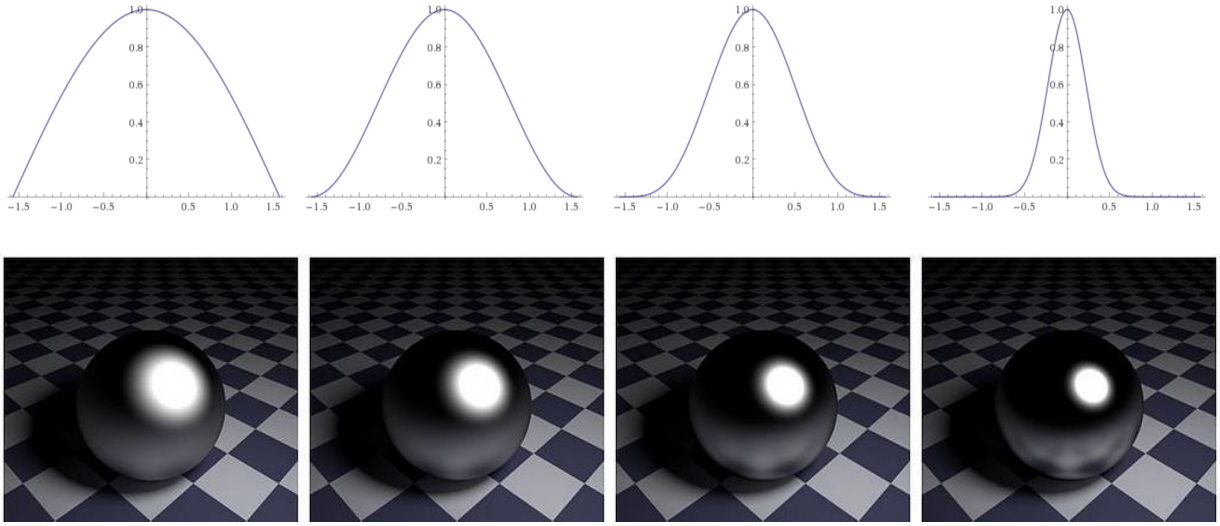
Phong

The so-called Phong shading model was proposed by Bùi Tường Phong. The model combines diffuse reflection with a *glossy reflection* and an *ambient color*, to simulate a broad range of materials. The full equation:

$$L = c_{ambient} + c_{diff}(\hat{N} \cdot \hat{L})l_{diff} + c_{spec}(\hat{L} \cdot \hat{R}_V)^{\alpha} l_{spec}$$

The model assumes that a material has a diffuse color c_{diff}, a specular color c_{spec} and an ambient color $c_{ambient}$. Typically, these colors are similar, since it looks strange when an orange has a blue specular reflection. Likewise, the light source can have a diffuse and a specular color, l_{diff} and l_{spec} respectively.

The color L that is reflected to the camera is the sum of a diffuse reflection $c_{diff}(\hat{N} \cdot \hat{L})l_{diff}$ and a glossy reflection. The glossy reflection is 1 when $\hat{L}$ is parallel to the view vector reflected in $\hat{N}$ (i.e., $\hat{R}_V$). When $\hat{R}_V$ it is not exactly equal to $\hat{L}$, the specular reflection is proportional to $(\hat{L} \cdot \hat{R}_V)^{\alpha}$, which drops quicker if α is larger:



Reflections

A rasterizer, unlike a ray tracer, cannot easily render arbitrary reflections. This is because a rasterizer has no access to global data: while rendering a single primitive, the other primitives are unknown, and can therefore not be reflected.

We can use tricks, however.

A classic trick is to duplicate the geometry behind a planar mirror. This also works for smooth water surfaces.

Another trick is environment mapping. We store the geometry surrounding an object in a texture, and map this texture to the object. Environment mapping has several limitations:

- The environment map has a finite resolution, so zooming in may not preserve all details.
- The environment map is static, unless we are willing to render the surroundings to it at the start of every frame.
- The reflective object cannot reflect itself.

Despite the limitations, environment mapping is still being used in virtually every game.

Normal Mapping

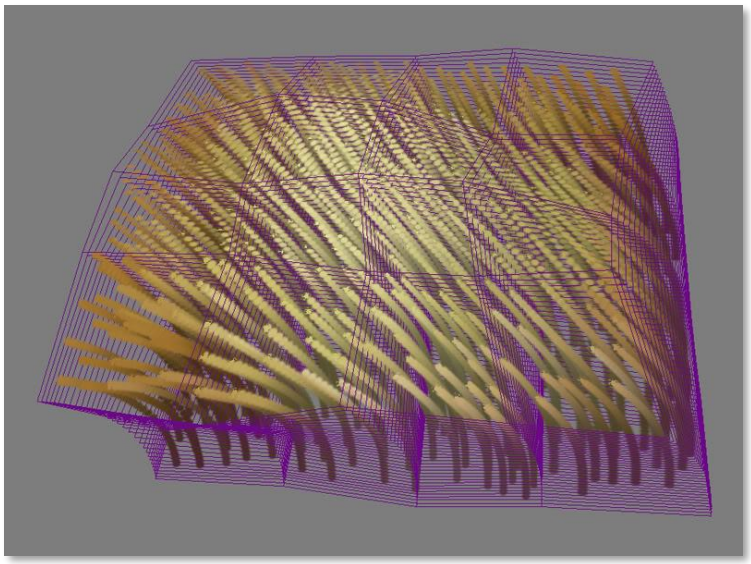
A normal map is a bitmap that stores normals in tangent space. Tangent space is a coordinate system at the location of a point we wish to shade: in tangent space the $\hat{z}$ vector is 'up', while $\hat{x}$ and $\hat{y}$ are vectors parallel to the surface. Once we have the three axii of tangent space we can transform the normals stored in the normal map to world space. The result is very fine control over local normals:

Note that the detail is not actually there in the geometry: we merely modify the local normal, like we also did with normal interpolation. Since the normal is used for reflections and shading, the added detail can be very convincing.

Fur

Rendering furry animals at first sight requires very complex geometry for the hairs. To overcome hardware limitations, game developers used a clever trick on the Sony Playstation 2 in *Shadow of the Colossus*.

The game uses a series of shells for the geometry of the creatures, and textures these shells with a special texture:



The texture contains a dot at the location of each hair. By layering the texture we get the illusion of hairs, although we can still somewhat see the gaps between the layers.

It turns out that even a small number of shells yields convincing results. This makes the technique applicable even on low-spec hardware such as the original Wii.

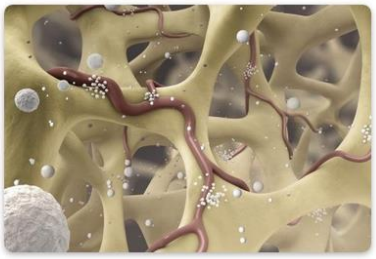
More Shading

Obviously the techniques described in this installment merely scratch the surface. You may want to look into steep parallax mapping, microfacet models, techniques for handling translucency, tricks for handling many lights and simulating fog. That last one is actually a post-processing technique, which we will discuss later in this course.

THE END

That’s it for this installment. If you have any questions, feel free to ask by email or on Slack!





INFOGR2019 – NEEDTOKNOW