A Procedural Shader Language for Ray-Tracing Applications

Kristoffe Zehr

Advanced Computer Graphics final project

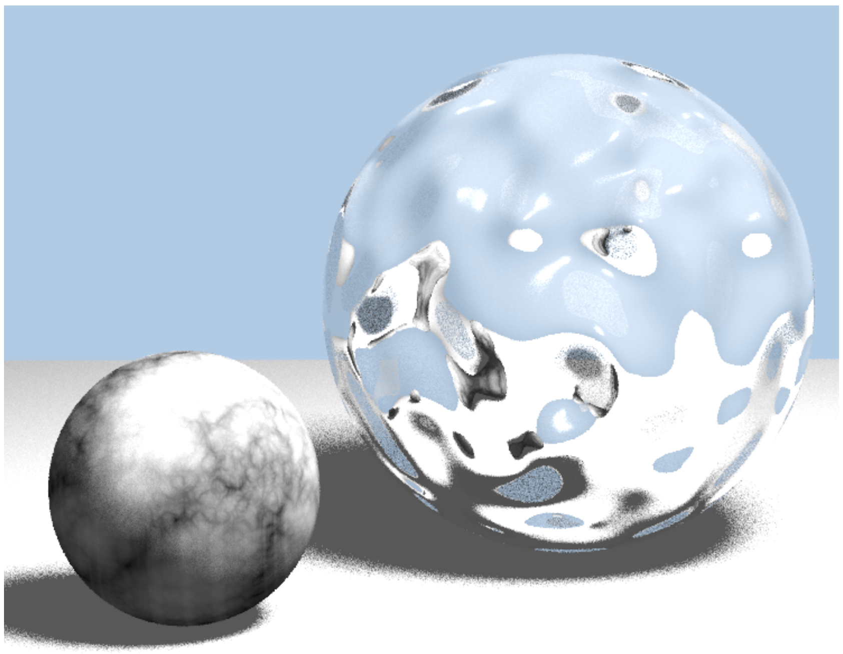

Figure 1: An image rendered using shaders, demonstrating procedural textures and bump mapping.

Abstract

Describes the implementation of a simple language for describing shaders using procedural graphics for a simple ray-tracer. Including color textures and bump-maps. The ray-tracer and procedural graphics functions are implemented in C++, and are called using Lua scripts.

1 Introduction

Procedural graphics is the process of generating 3D models or textures using mathematical functions rather than storing them in memory. It has several advantages over traditional methods of storing graphics. Firstly, textures can be very high resolution without the need to store a large texture in memory, since it is generated on the fly. A texture function takes in a position as an argument and returns a scalar or color value, so there is no need to map a texture to a complex surface. This eliminates distortion and artifacts on the texture.

Traditionally, procedural shaders have only been used in real-time applications. GLSL is one example of a shader language which can create textures and bump-maps in real time using the GPU. The goal of this project is to apply the advantages of procedural shaders to the more accurate rendering of a ray-tracer to create very interesting images.

2 Previous Work

This project makes heavy use of the simplex noise function described by Ken Perlin. N-dimensional noise can be determined by mapping pseudo-random gradients to each integer point on a grid, and then interpolating the values of the 2^n closest gradients. This algorithm is described in [Perlin 1985] and then improved upon in [Perlin 2002].

In addition to the simplex noise function, this project also makes use of the turbulence function described in [Perlin 1985]. Which takes

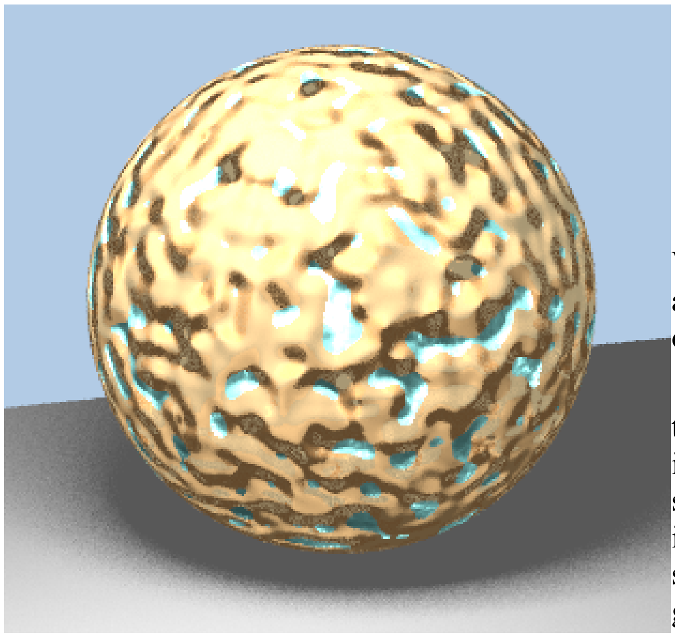

Figure 2: An example of a bumpy copper texture with corrosion in the "pits" of the bump map.

the sum of multiple noise functions. It is described by the equation:

$$\sum\left(\frac{noise(p*2^{i})}{2^{i}}\right)$$

Where p is a vector representing the position of the point in space, and i=1,2,3,...n. Where n is the number of noise functions to sum.

3 Implementation

In order to ensure that the shader language does not cause significant performance drops, the ray tracer and the procedural functions were both implemented in C++.

3.1 Bump Mapping

In order to implement bump mapping, I decided to use a shader function which takes in a vector representing a point in 3D space and returns a scalar value representing the “height” of that point. A procedure written in the C++ portion of the program then calls this shader function 6 times. Each time off-setting the position along one axis and determining the gradient for that axis. The equation for the new normal offset u for a given point [x,y,z] is:

$$\frac{height([x+\epsilon,y,z])-height([x-\epsilon,y,z])}{\epsilon},$$
$$\frac{height([x,y+\epsilon,z])-height([x,y-\epsilon,z])}{\epsilon},$$
$$\frac{height([x,y,z+\epsilon])-height([x,y,z-\epsilon])}{\epsilon}$$

Where ε is a very small value by which the points are offset. The final normal vector is then the original normal minus the unit vector of u.

The advantage to using a height map rather than simply generating random unit vectors and interpolating their values is due to the fact that the simplex noise function is deterministic. For any input, the noise function will always return the same output. Therefore, the height map used when generating a bump map can be calculated in other parts of the shader program to create effects such as corrosion on metal or oceans when generating a planet texture.

3.2 The Shader Language

The decision of what language to use when writing shaders had several important factors. Firstly, the language must be extremely fast so that there is minimal decrease in performance of the ray-tracer. Secondly, it must be easily integrated with C++ so that it can call C++ functions and be run as a script from C++ programs with minimal overhead. Finally, it must be a simple language that is very easy to program in.

The language which I decided on using is Lua. Lua is a fast, simple scripting language written to the ANSI C standard. Therefore, it is easy to integrate into C and C++ applications with the provided API. It is one of the fastest, if not the fastest, scripting languages provided for C, and is extremely simple and powerful.

In order to integrate Lua with C++, the functions callable from Lua must be written. Any function that will be called from Lua must return an integer and take only one parameter, a pointer to a lua_State struct. In order to access the parameters, the function must pop them off the top of the Lua stack. After calculations are completed, the function returns a value by pushing it onto the Lua stack[Ierusalimschy 2006].

Since Lua is written in ANSI C, it does not always support dynamic linking of libraries. Therefore, the functions written in C++ must be pushed onto the Lua stack before any scripts are run.

In order to specify what materials use shaders in a mesh, the .obj file parser was updated. When defining a material in an .obj file, the prefixes “texture_shader”, “bump_shader”, and “reflect_shader” can be used to define the filenames of the diffuse color shader, bump map, and reflective color shader, respectively. These scripts are then loaded into the program and pushed onto the Lua state. When the ray-tracer needs to retrieve a normal vector or color value, it pushes the position onto the Lua stack and runs the appropriate function, then retrieves the value by popping it off of the Lua stack[Ierusalimschy 2006].

4 Results

With the implementation of the shader language, the ray-tracer does not show any significant performance drops except when rendering shaders with bump maps, as the height map needs to be calculated 6 times for every ray that intersects the surface. However, this is at worst a constant drop in performance, with no additional drops due to the complexity of the height map function.

Using only the functions coded in the application and the math functions provided in Lua, many complex shaders can be developed. The ray tracer allows the bump mapping effects to be improved by adding realistic reflections to the bumpy surface.

5 Conclusion and Future Work

Possible extensions to this project could include adding improvements to the ray tracer such as refraction and caustics, which would improve the images greatly. Due to the procedural nature of the textures involved, a radiosity solution would be difficult to implement, since calculating the average color of a patch would be a very complex process. This could be avoided by simply allowing the user to define an average color for the shader.

Additionally, adding more procedural functions such as Voronoi diagrams or a brick texture could improve the variety of shaders that could be created. The ray-marching method of hypertextures is also very well suited for implementation in a ray-tracer. All that would be necessary is modification of the hit detection code. Hypertextures would not only improve the appearance of a shader, but would also effect the shape of the shadow cast by an object. Things such as organic shapes in rock or clouds could be easily implemented procedurally.

6 References

Ierusalimschy, Roberto. *Programming in Lua*. Rio De Janeiro: Lua.org, 2006. Print.

Ken Perlin, *An Image Synthesizer*, ACM SIGGRAPH Computer Graphics, v.19 n.3, p.287-296, July 1985

Ken Perlin, *Improving Noise*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, p.681-682, 2002