An introduction to ISL

for them not too shy (WIP Update 4)

Table of Content

Foreword

The purpose of this booklet is to lead the novice, supposedly impressed by the notion of coding, into a guided first contact with the **Indigo Shading Language** (ISL).

As such, the reader is expected to have been through UV texturing and is now willing to use a different method to define materials in Indigo.

From there, the tutorial will try to conceal the technicity that ISL may involve by proceeding from one illustrated step to another. Sometimes in an imaged language, often at the price of technical accuracy. Theoretical insights will happen when required.

Text in coloured fonts are WIP artefacts.

Introduction

We will progressively be building the simplest **2d shader**, then add to it a couple of UI controls.

A 2d shader, such as a procedural **noise**, will cover a surface like a texture does: according to the surface UV coordinates[1], as they were projected onto the mesh. Therefore, all 2d shaders are functions of the U and V grid.

Start **Indigo** (Indigo **RT** all the same) standalone application.

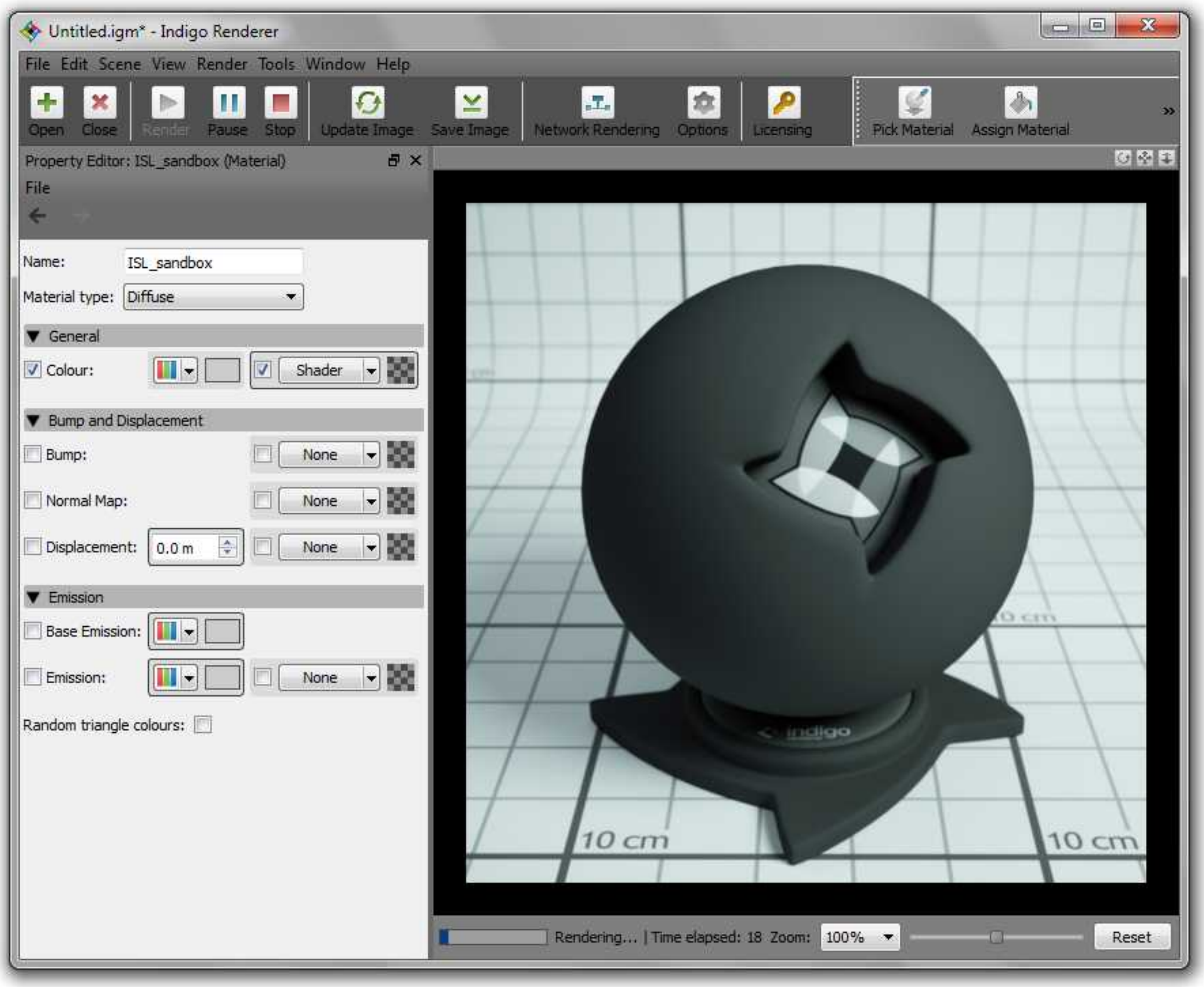
1. These coordinates will be used by Indigo in order to locate the point at wich the light response must be evaluated.

Initial stages

Open the **File** menu and create a **New Material**.

Make sure that the **Property Editor** is visible. It is otherwise called from the **Window** menu. Also, from a contextual menu available by right-clicking on the title of another dockable panel.

My own working setup looks like this:


Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Property Editor: ISL_sandbox (Material)
File
General
Bump and Displacement
Emission
10 cm
10 cm
indigo

Name: ISL_sandbox
Material type: Diffuse
Colour: ✓ Shader ✓
Bump: None
Normal Map: None
Displacement: 0.0 m None
Base Emission:
Emission: None
Random triangle colours:
Rendering... | Time elapsed: 18 Zoom: 100% Reset

The only panel that we will be using is the Property Editor.

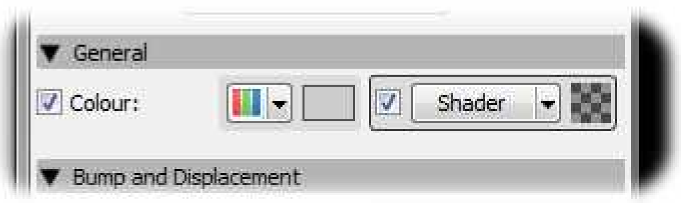
The Property Editor: a quick overview

The Property Editor is the Indigo object editor.

Give the material a **Name**. We will then have a look at the Property Editor layout titled **General**.

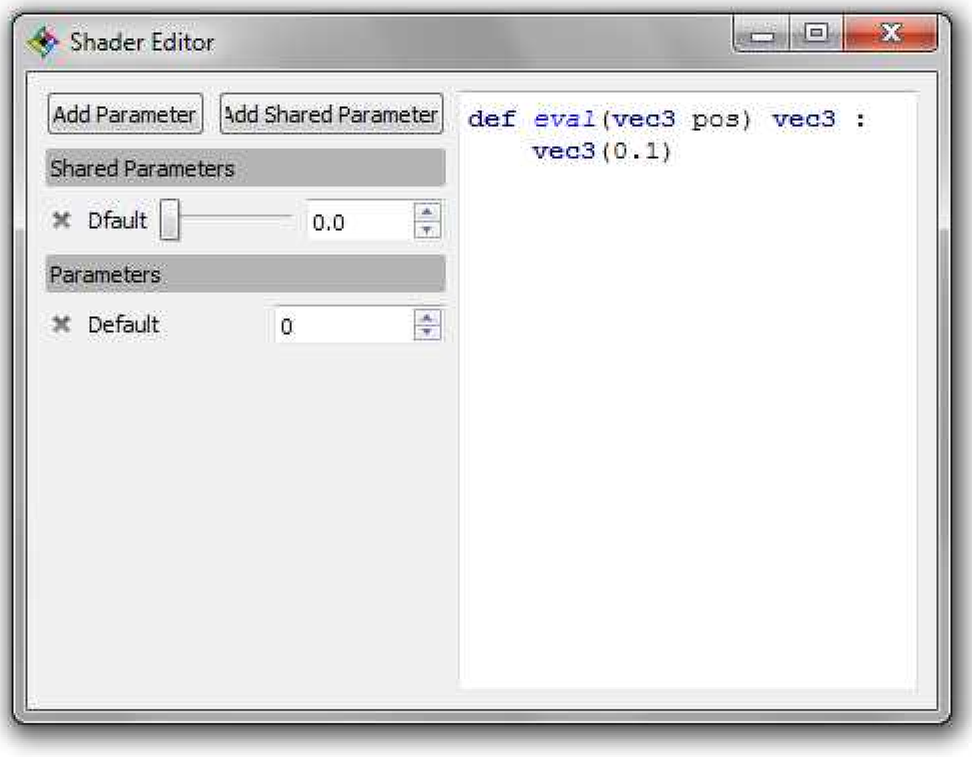
For a few versions, new materials in Indigo standalone come with a shader for *colour*. On semantics: *colour* is a material parameter (aka: channel, attribute), and *shader* is a type for a parameter. In other words one can say that the current type of the colour parameter is: shader.

Click on the checker icon on the right of the pulldown menu in order to open the editor fitting with the current parameter type:



The general section of the Property Editor. Material types other than diffuse may expose more controls.

Now look at the text field only:

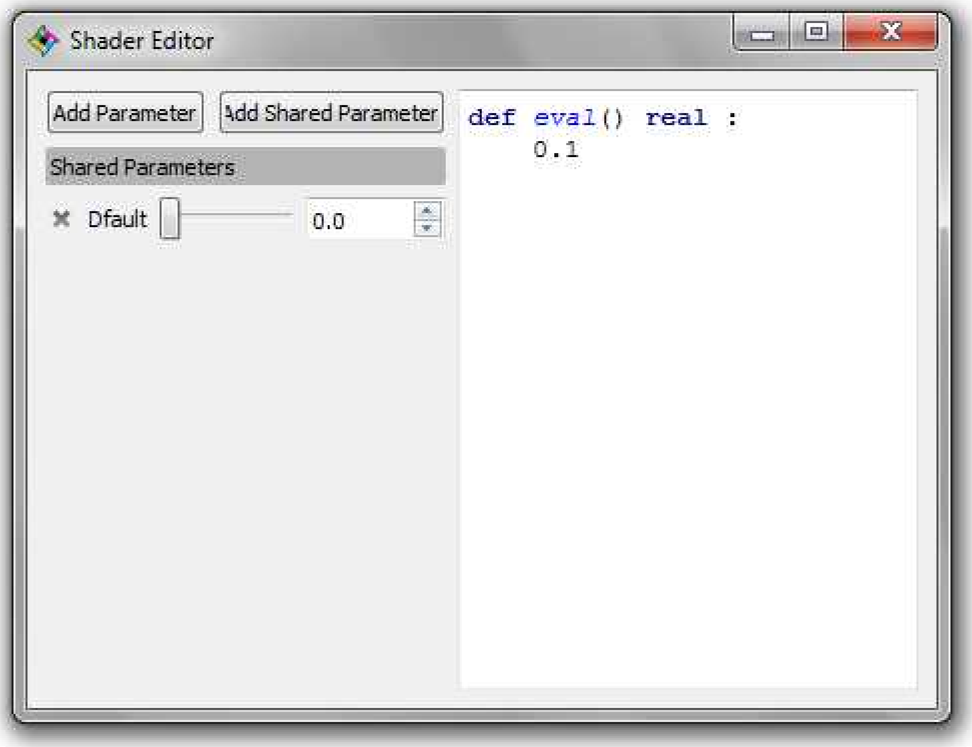


```
def eval(vec3 pos) vec3 :
    vec3(0.1)
```

The colour shader editor.

Ask yourself one thing: does it look like scary, knowing that you don't have to be concerned by the first line ? I mean, at all.

If yes, change now the *bump* parameter type to *shader* and let's have a look at this one. Ignore the first line, ignore the dummy control on the left and don't kid me...



```
def eval() real :
    0.1
```

The bump shader boils down to a single floating point value.

Granted, the result is unimpressive but it's a valid shader code already. Mark - my - words:

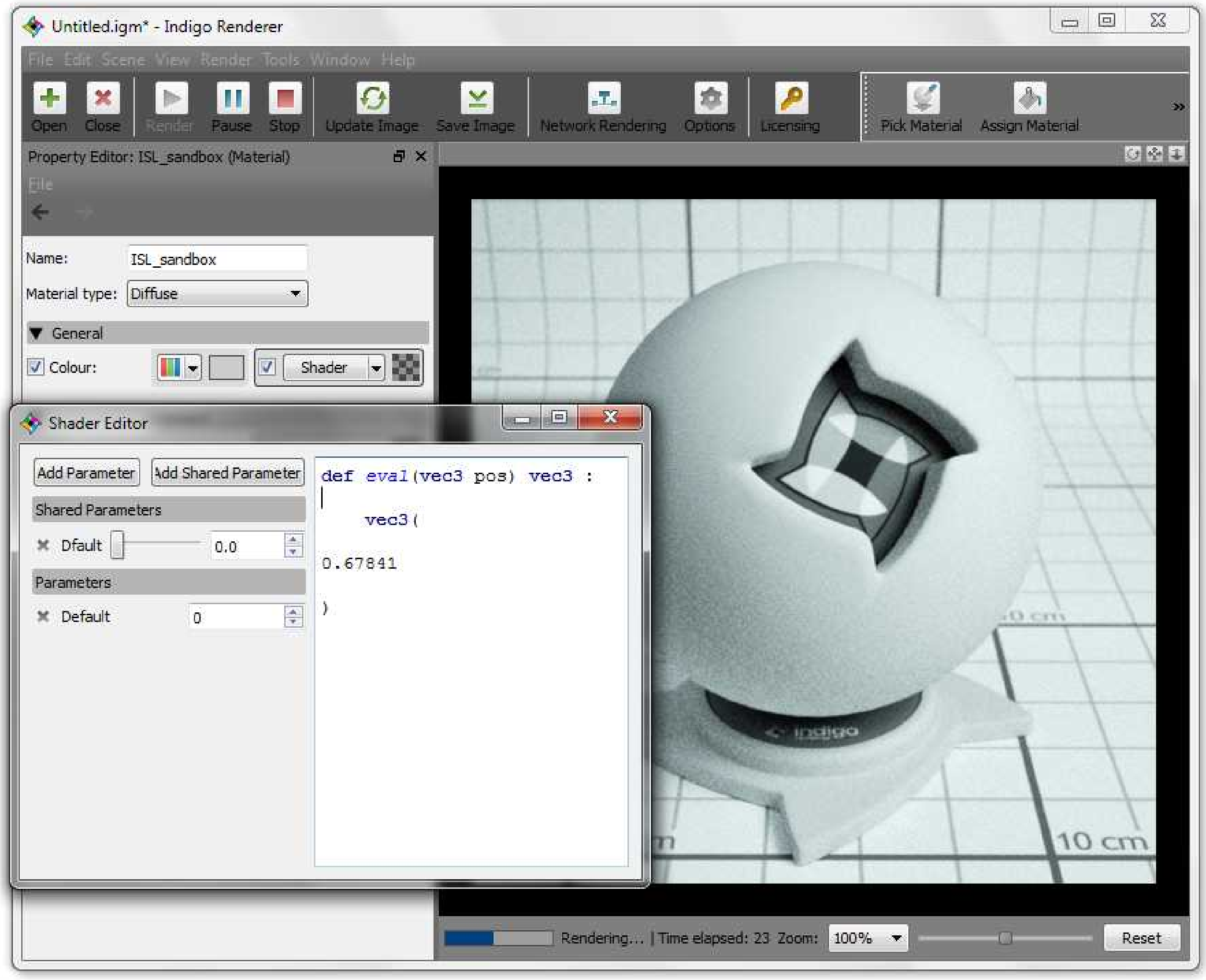
A single floating point number applied at a specific UV coordinate makes a procedural bump shader, and you can make it happen.

Thank you for your attention.

Right, turn off the bump mapping evaluation for now by ticking off the leftmost checkbox in the bump parameter controls.

We will be using the colour parameter for a better visual feedback while laying out the basics. Go back to its shader editor and play with the value within the parenthesis in the second line.

editing tip: put the value alone on a new line, with the last bracket below. Editing the value is more practicable this way.


Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Property Editor: ISL_sandbox (Material)
File
Shader Editor
Add Parameter
Add Shared Parameter
Shared Parameters
Parameters
Rendering... | Time elapsed: 23 Zoom: 100%
Reset
10 cm

Name: ISL_sandbox

Material type: Diffuse

General

Colour: Shader

Dfault 0.0

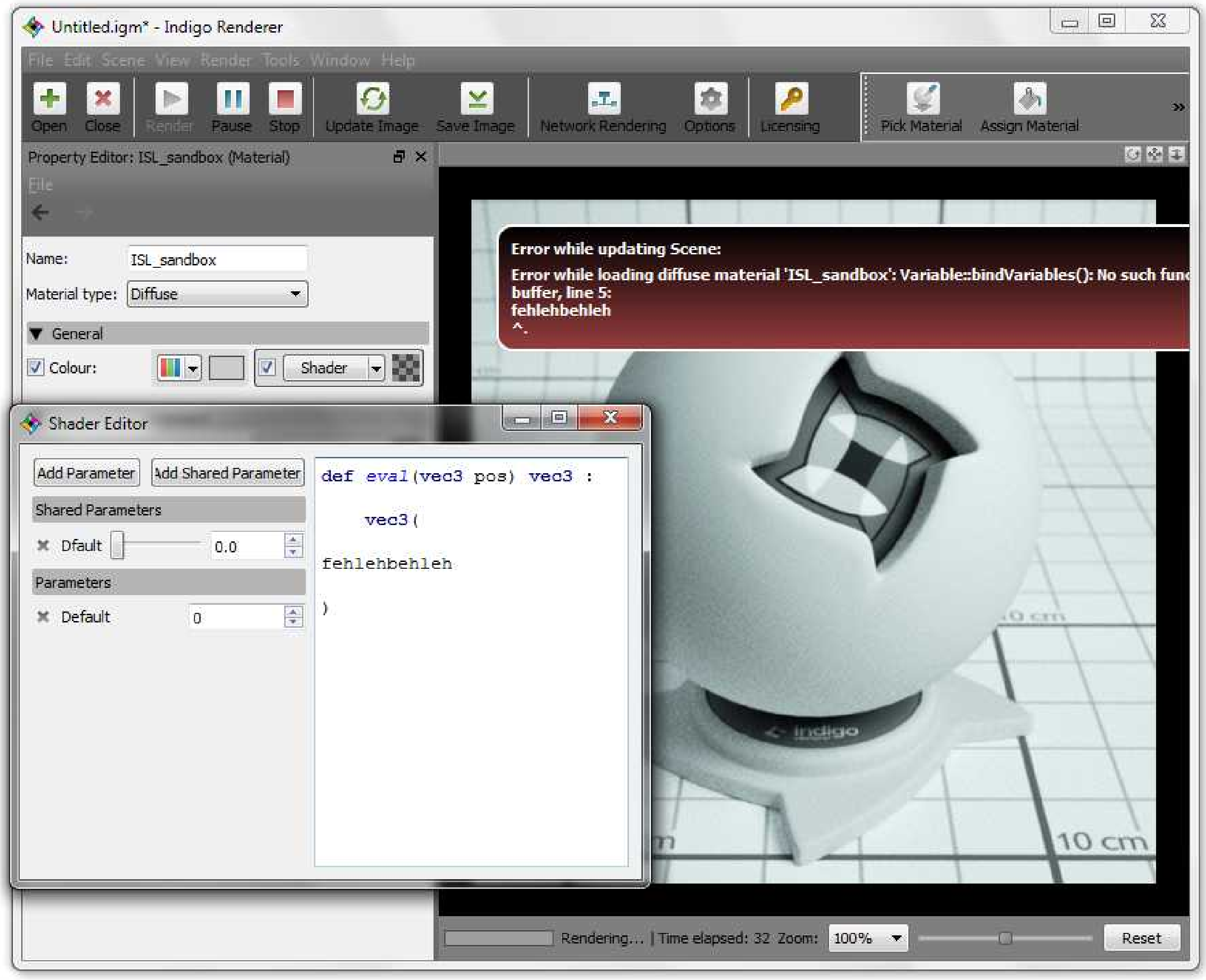
Default 0

```
def eval(vec3 pos) vec3 :

    vec3(

0.67841

)
```

Tweaking the default value of the colour shader.

The ISL debugger

While editing the *colour* shader value, you may have noticed that Indigo would reject the code between two valid inputs. This is signified in two ways: the shader has not reflected any change compared with the last valid input, and there is an explicit report overlay in the upper render panel region.


Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Property Editor: ISL_sandbox (Material)
File
General
Shader Editor
Add Parameter
Add Shared Parameter
Shared Parameters
Parameters
10 cm
Rendering... | Time elapsed: 32 Zoom: 100%
Reset

Name: ISL_sandbox
Material type: Diffuse
Colour: Shader
Dfault 0.0
Default 0

```
def eval(vec3 pos) vec3 :

    vec3(

fehlehbehleh

)
```

```
Error while updating Scene:
Error while loading diffuse material 'ISL_sandbox': Variable::bindVariables(): No such func
buffer, line 5:
fehlehbehleh
^.
```

You know when you did something wrong, and generally what it is.

You will eventually get used with the different types of errors identified, at first confusing the ISL debugger is of a great help nonetheless.

Be not troubled though, the ISL debugger happening while editing a shader code is something casual.

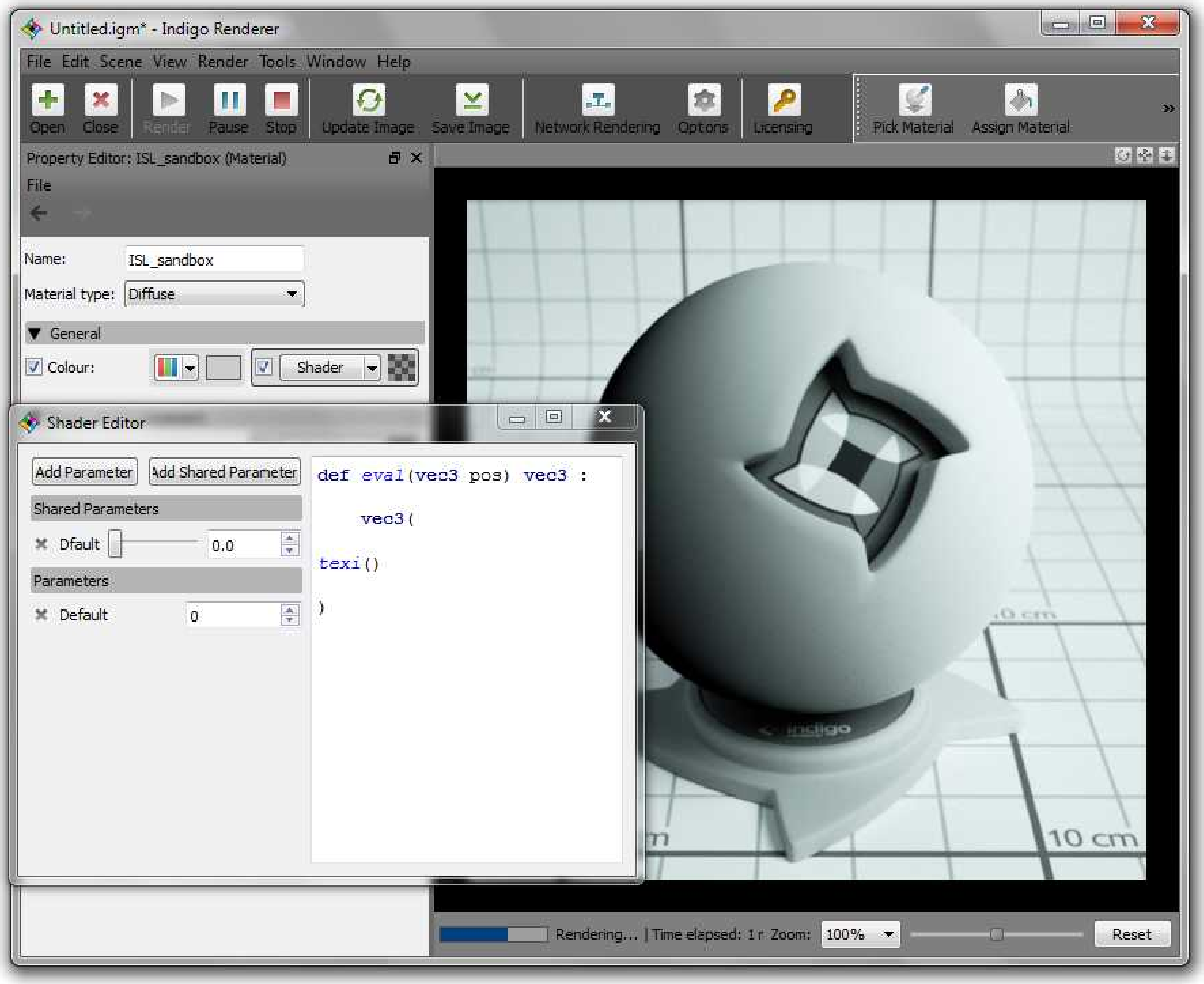
By the way, don't assume that *fehlehbehleh* can not make it into ISL. It can.

Gray is boring

And there is more where it came from ! How about gradients now ?

Remember when I said that a shader was a function of, notably, the U coordinate and that it sounded scary already ?

Well, let's expose the U coordinate of the material test sphere with no further discussion:


Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Shader Editor
Add Parameter Add Shared Parameter
Rendering... | Time elapsed: 1 r Zoom: 100%
Reset

Property Editor: ISL_sandbox (Material)

Name: ISL_sandbox

Material type: Diffuse

General

Colour: Shader

Shared Parameters

Dfault 0.0

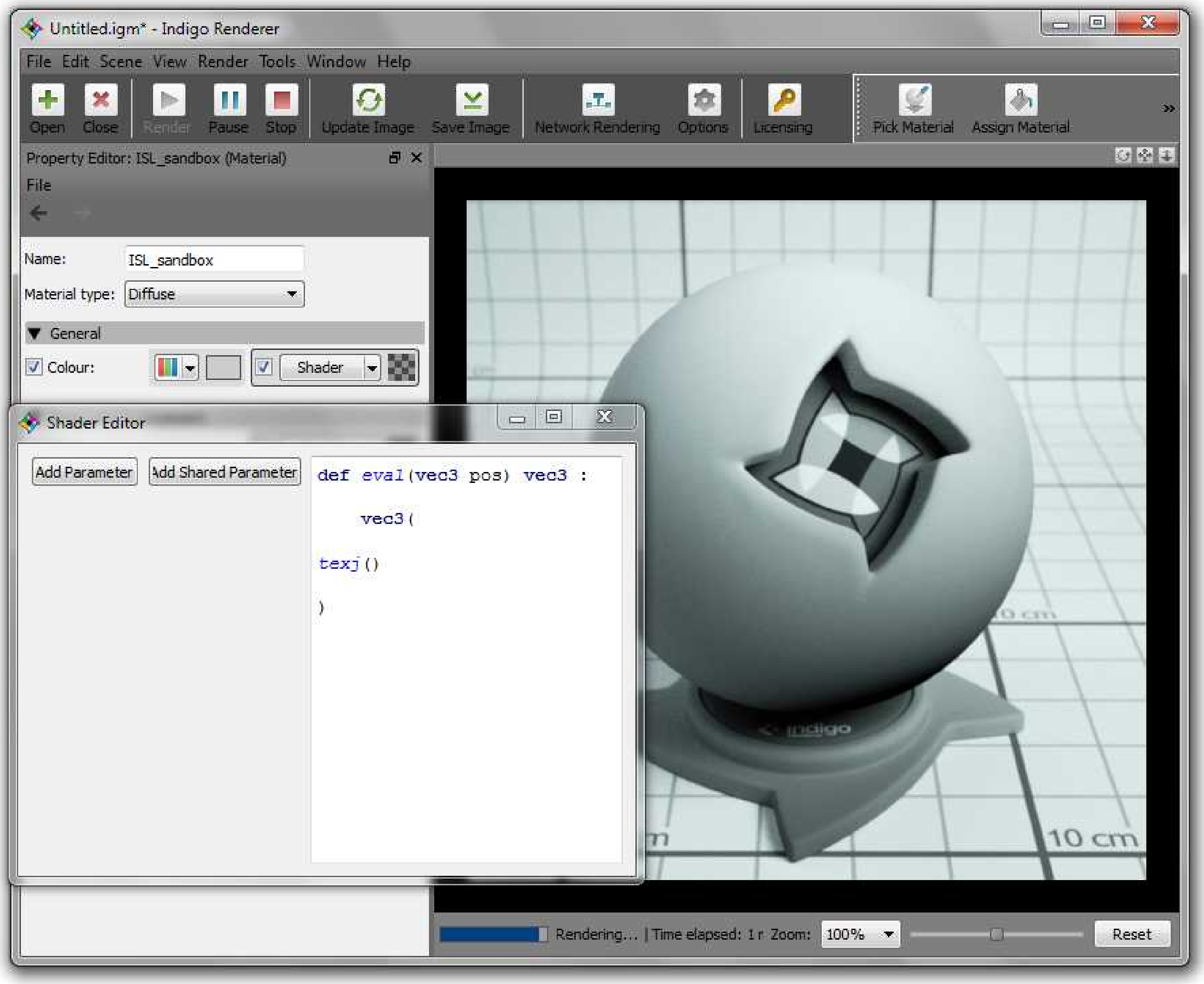
Parameters

Default 0

```
def eval(vec3 pos) vec3 :

    vec3(

texi()

)
```

The U coordinate exposed.

texi() is one name of the U coordinate in Indigo Shading Language, *texj()* goes to the V coordinate. Both apply only to the first UV set, as the shortcuts (aliases) they are.



Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Property Editor: ISL_sandbox (Material)
Shader Editor
Add Parameter
Add Shared Parameter
Rendering... | Time elapsed: 1 r Zoom: 100%
Reset

Name: ISL_sandbox
Material type: Diffuse
General
Colour: Shader

```
def eval(vec3 pos) vec3 :
    vec3(
texj()
)
```

The V coordinate of the test scene geometry.

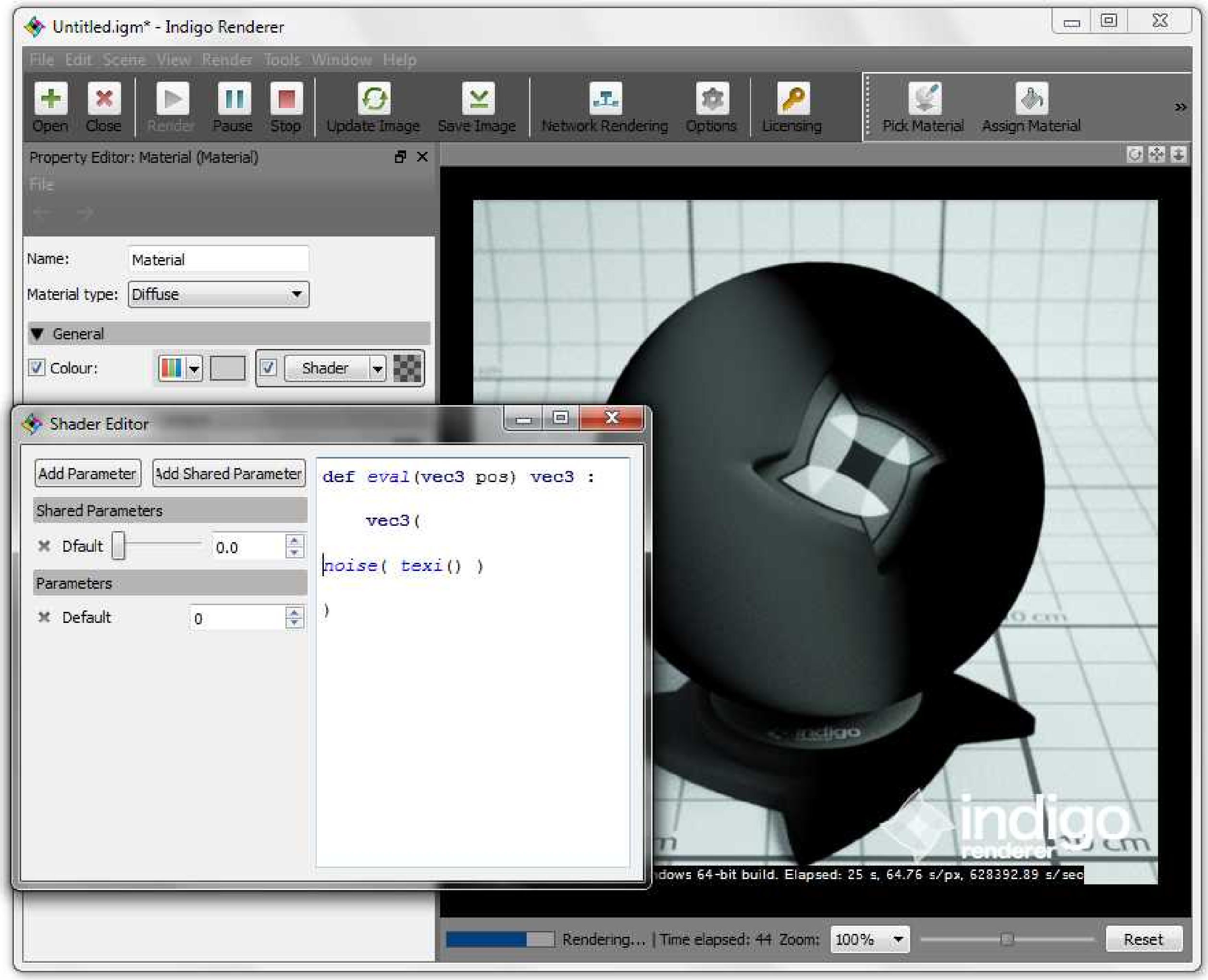
Now can you see the damn UVs oooooh boooy... oh sorry, my memories with ISL...

In this chapter, **we have exposed the data that Indigo will be using to lay out 2d shaders.**

Drawing noise with a ruler

Thanks to computers we don't have to. Give one a ruler and it will draw the noise out of it. And guess what: *texi()* and *texj()* make two rulers, we have got choice for this part coming.

So, the noise takes a ruler and the ruler is, say, *texi()*. Repeat after me: "*noise()* takes a ruler and the ruler is *texi()*". Goood.



Name: Material
Material type: Diffuse
General
Colour: Shader

Dfault 0.0
Default 0

```
def eval(vec3 pos) vec3 :

    vec3(

noise( texi() )

)
```

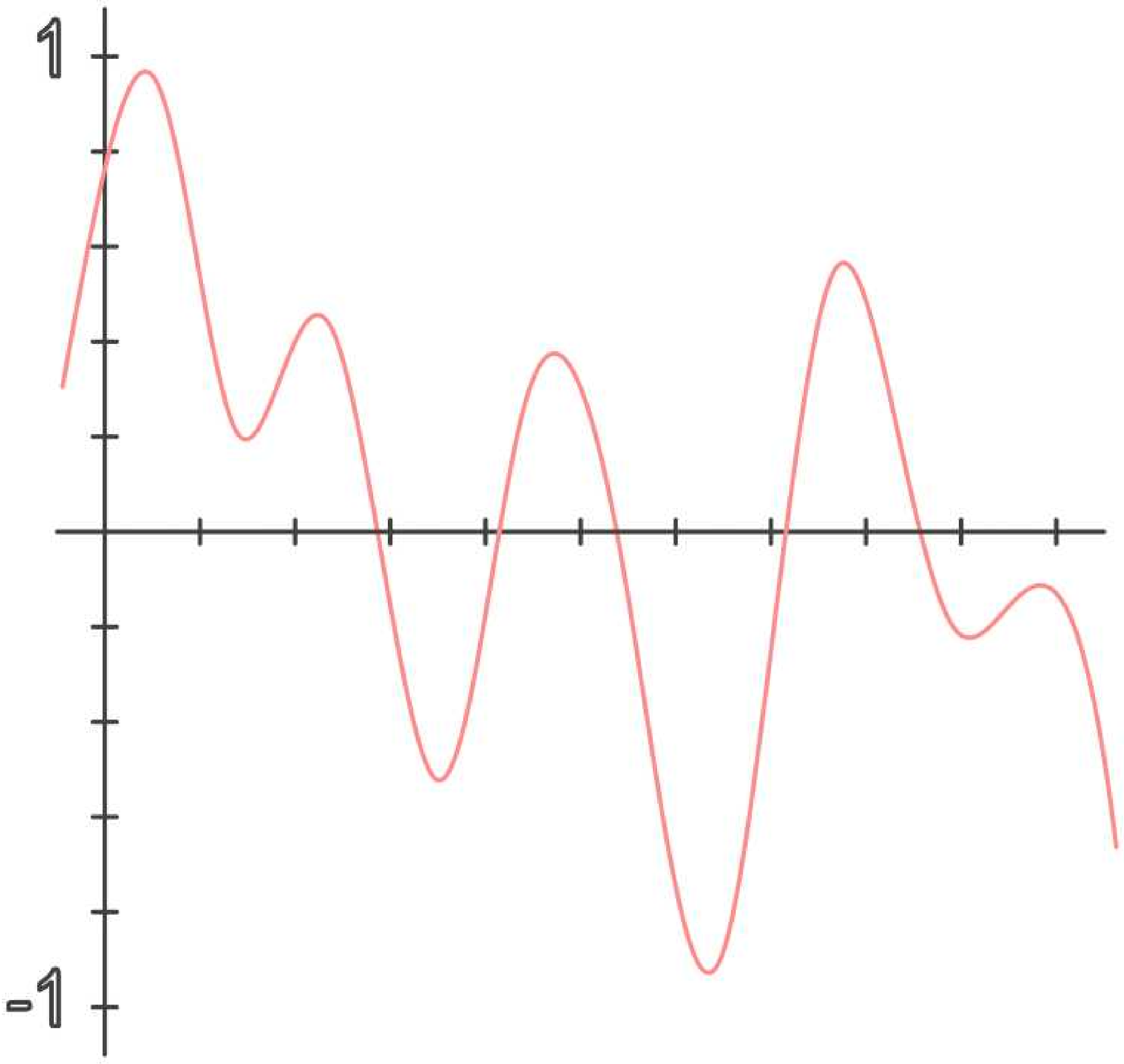
dows 64-bit build. Elapsed: 25 s, 64.76 s/px, 628392.89 s/sec

Zoom: 100%

You have been hypnotized.

A note on colour and functions

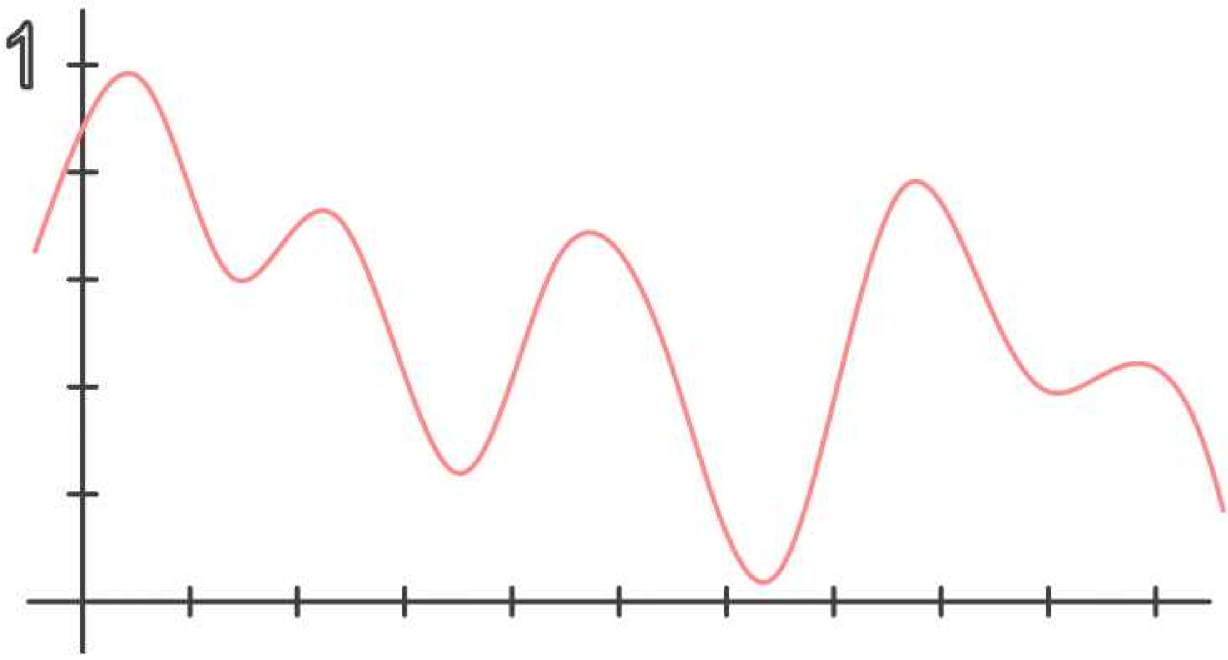
It is a property of many functions to oscillate between minus one and one. It is however a property of colour to be bound between zero and one. Specially, these colour attributed to an **albedo**, ask wikipedia. This is easy to understand: no diffuse material can be darker than black, nor lighter than white. In fact, a sheet of standard white paper has a measured albedo of 0,85; you should consider this as a hard limit for an albedo value.



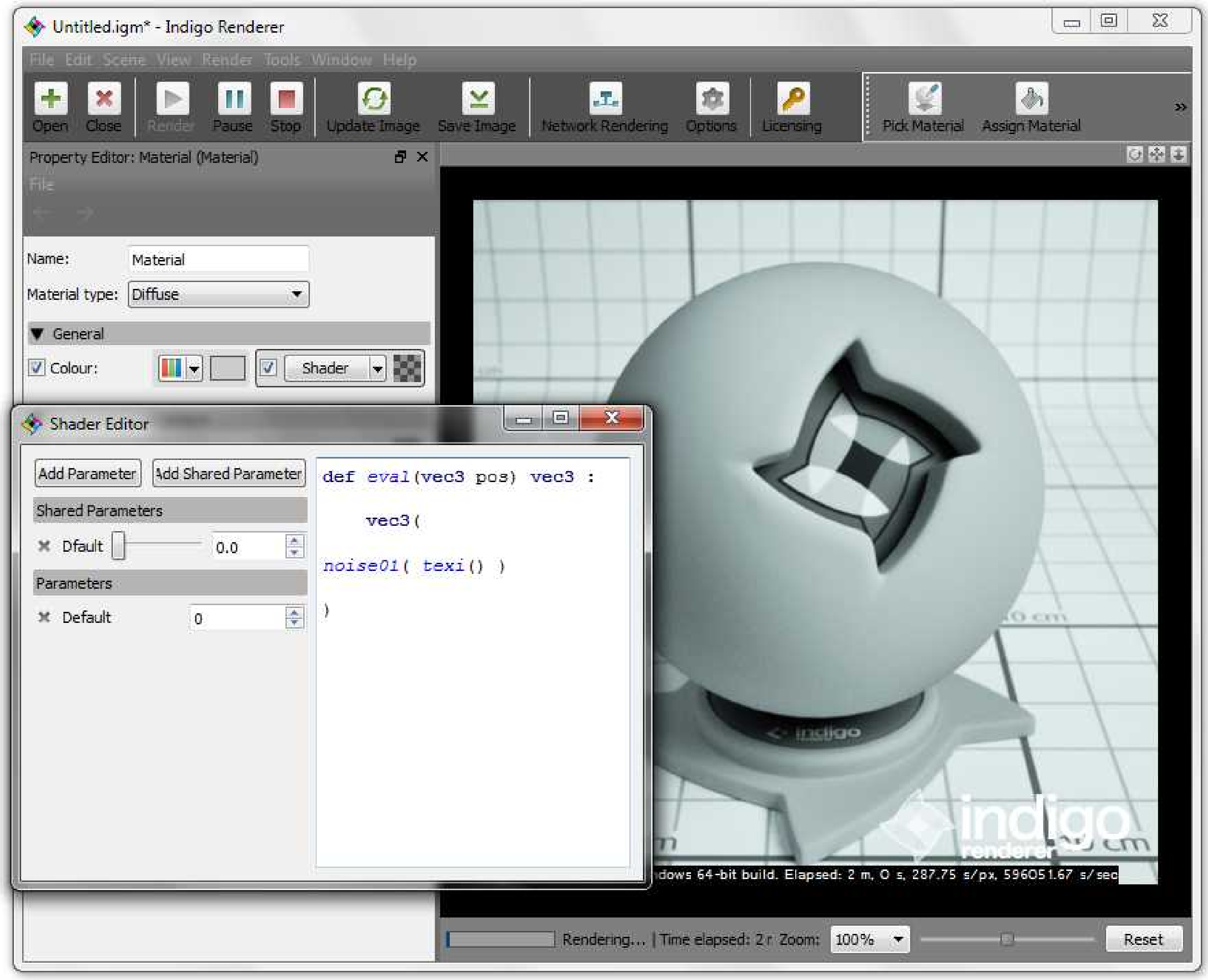
Noise functions are generally oscillating between minus one and one.

This property of the *noise()* function, added to the one of an albedo explains why a good half of our material is pitch black: it is the domain of the *noise()* function that equals, or is below zero.

This is easily adressed by replacing *noise()* with *noise01()*, a variant having its output scaled in order to fit entirely between zero and one.... hence the name !



noise01() *is bringing the entire output of* noise() *into the albedo domain.*



Name: Material
Material type: Diffuse
Colour: Shader
Dfault 0.0
Default 0

```
def eval(vec3 pos) vec3 :

    vec3(

noise01( texi() )

)
```

dows 64-bit build. Elapsed: 2 m, 0 s, 287.75 s/px, 596051.67 s/sec

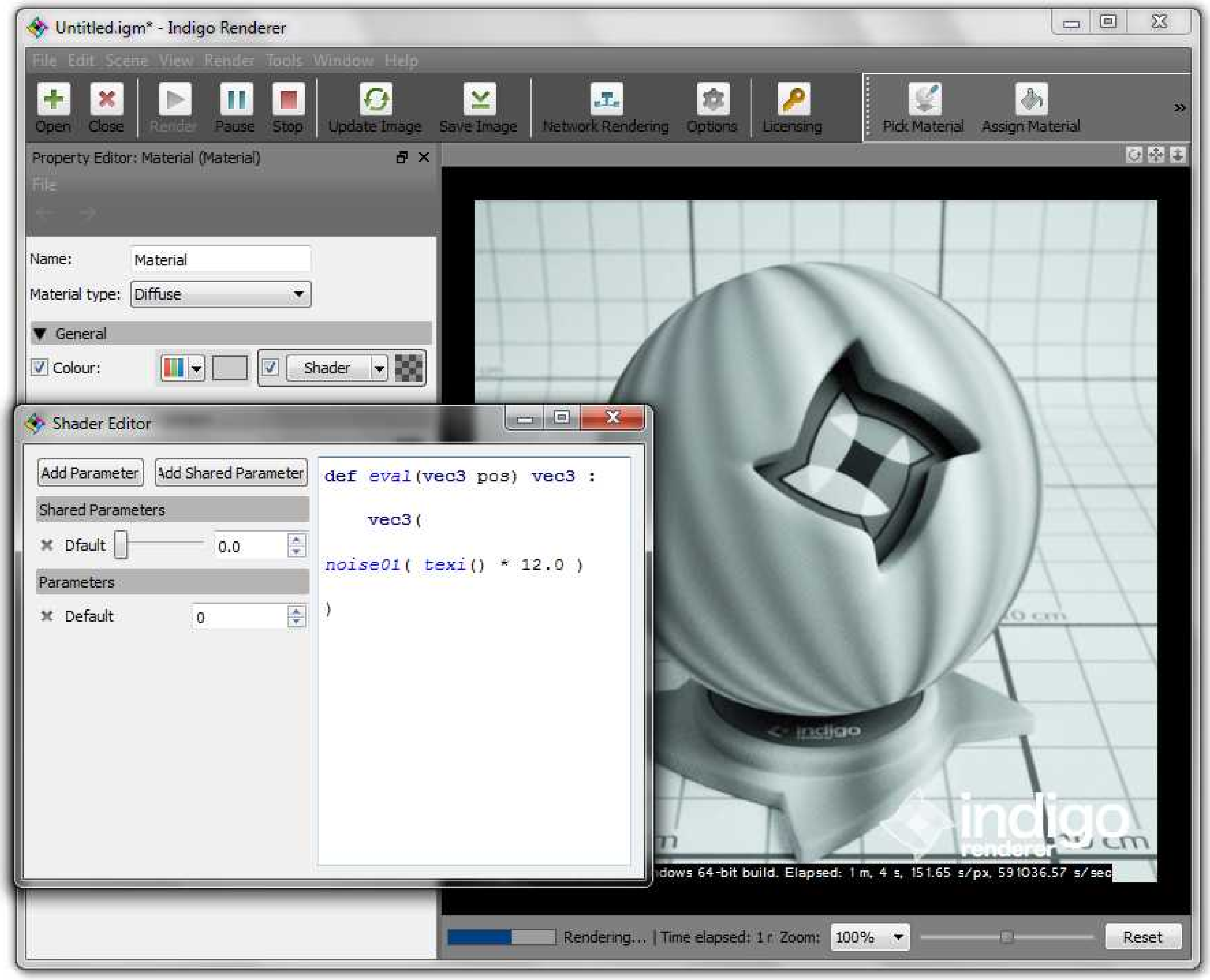
Rendering... | Time elapsed: 2 r Zoom: 100%

Scaling the ruler

Now that we can see something (do we) let's see it even better. If you are an artist I will not be teaching to you that for this purpose, one need to step back off the subject.

In computer shading, one would make the ruler smaller so more of them rulers could fit along the surface. If I **multiply** the readings of *texi()* (the ruler) by two I will reach 1.0 twice as fast over the surface, and such a second ruler would fit along.

Let's multiply *texi()* by two. No wait, it's not conclusive. Make it twelve, *ie* 12 **dot** 0. The terminal number is optional, but the period is not.


Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Property Editor: Material (Material)
File
Shader Editor
Add Parameter
Add Shared Parameter
Shared Parameters
Parameters
Reset

Name: Material
Material type: Diffuse
General
Colour: Shader

Dfault 0.0
Default 0

```
def eval(vec3 pos) vec3 :

    vec3(

noise01( texi() * 12.0 )

)
```

dows 64-bit build. Elapsed: 1 m, 4 s, 151.65 s/px, 591036.57 s/sec

Rendering... | Time elapsed: 1 r Zoom: 100%

*The value DOT something, because **12** and **12.0** are not of the same nature to Indigo.*

Yes, yes, yeees it's happening. You are getting in control, and this achievement will be sanctioned.

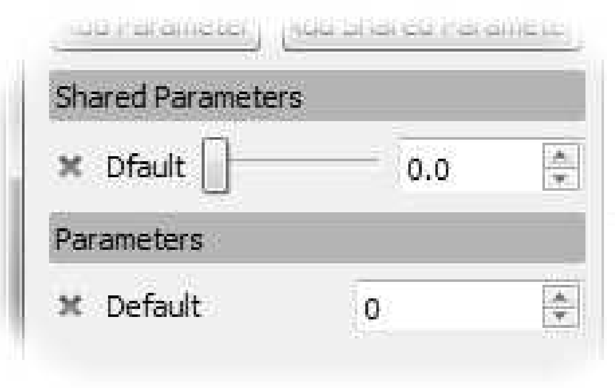
Params

Scaling UV readings as we just did is a primary feature for a procedural shader. Unless you know exactly the dimensions of the geometry it will be applied to, you want to make this property readily available for tweaking the shader, in order to have it fit under varied sceneries.

This is the purpose of **params**. Not to be confused with material parameters as introduced previously, a param is a custom shader parameter, wich translates into a shader editor control.

You most certainly noticed the default params already there. While we could use the first of them in order to scale the *vec3()* colour output for instance, we will rather go through creating the one adressing our former concern, with an adapted output range.

Then should you be able to scale the *vec3()* term as suggested. All by yourself, that would be a good exercise.



Shared Parameters

Dfault 0.0

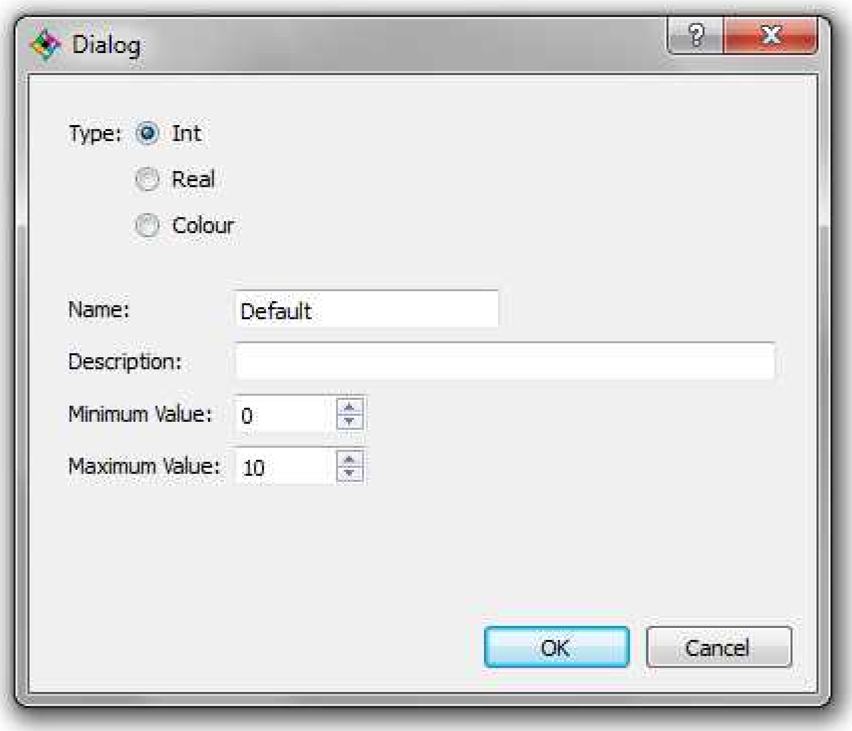
Parameters

Default 0

The default params in the colour shader editor.
They could have been deleted from the start.

Delete each param from the colour shader editor by clicking on its leftmost **x** icon, then confirming the param deletion. Indicatively, you should find yourself with an editor as seen in the image titled "*The V coordinate of the test scene geometry*". link

Click then on the **Add Shared Parameter** button in order to summon the window for param creation.



Type: [✓ Int] Real Colour

Name: Default

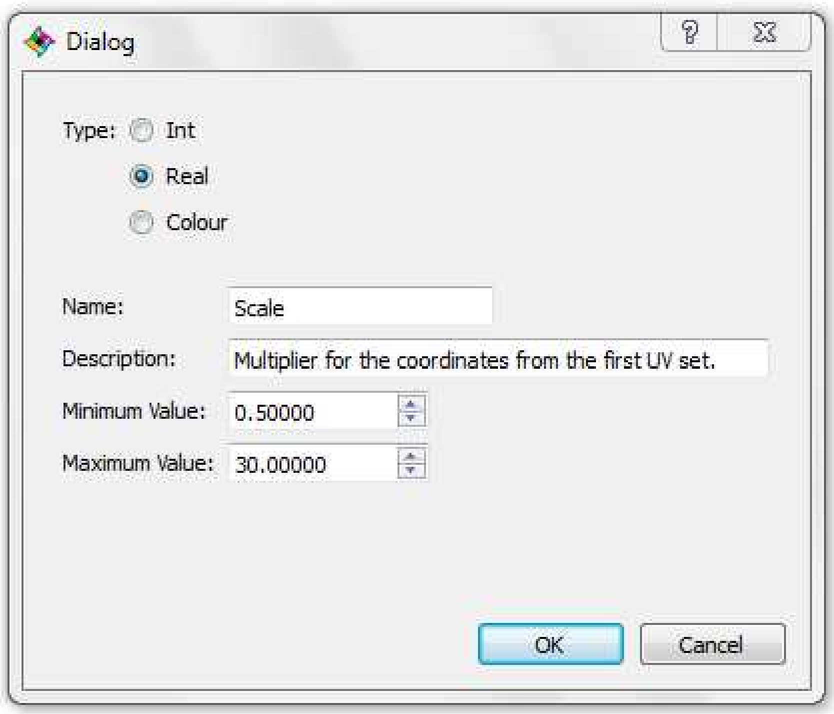
Description:

Minimum Value: 0

Maximum Value: 10

The param creation window is featuring several captions that define the param to be created: **Type**, **Name**, **Description**, **Minimum Value** and **Maximum Value**. All are self-explanatory but perhaps the first one; data types concerns though will be insistently eluded during this introduction to ISL.

Change the param creation options as shown below:



Type: Int [✓ Real] Colour

Name: Scale

Description: Multiplier for the coordinates from the first UV set.

Minimum Value: 0.50000

Maximum Value: 30.00000

param creation options setup for Scale.

The type *must* be **Real** for the operation being (*noise01()* delivers that data type).
The name will be **Scale** so we can stay sync'ed during the tutorial. Scale is not a keyword in itself.
The description is for your interest. It is a good practice to provide one.
The min and max values are at your discretion.

Click **OK** to create the *Scale* param.

At this point, all you will get is a new fancy way to restart the render. Not. Bad.

Note on params edition

Shader parameters can not be edited at this time, they must be replaced when needed. Several params with the same name may coexist; Indigo would currently ignore subsequent definitions.

Shared material data

It was not mandatory to pick a shared param to create, but UV scaling is that kind of effect that one wants often applied simultaneously to several parameters of a material. For instance, in order to maintain a correlation between its colour and bump parameters.

Ordinary params are visible only for the current shader parameter, while shared params are available to all shaders of a material; they will appear in each of its shader editors.

There exists also a mean to define shared ISL functions, currently by hand editing an Indigo file (IGM or IGS).

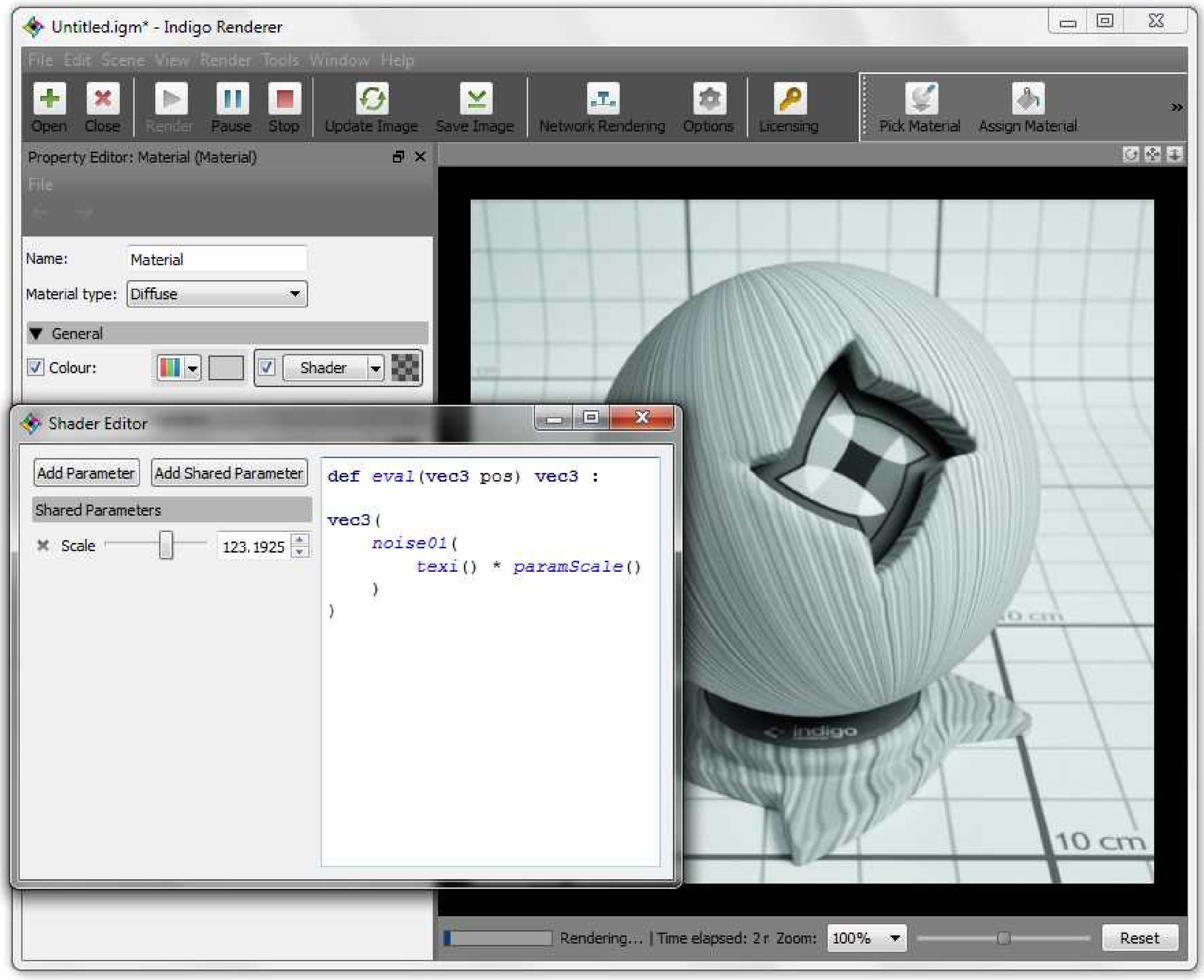
Effective control

It is finally time to bind the UI control to the ISL shader. While not obvious, the convention is easy to remember.

ISL will aknowledge the param with the user defined name **Scale** as *paramScale()*. This is why it is a good practice to give params a name with a capital first letter.

paramScale() will act as a float value in ISL, because its type was defined as real (both are synonyms). The parenthesis signify that ISL will interpret the param as a function, entity wich will be covered in a different work[2].

To arms, now.


Untitled.igm* - Indigo Renderer
File Edit Scene View Render Tools Window Help
Open Close Render Pause Stop Update Image Save Image Network Rendering Options Licensing Pick Material Assign Material
Property Editor: Material (Material)
File
General
Colour:
Shader
Shader Editor
Add Parameter
Add Shared Parameter
Shared Parameters
10 cm
Rendering... | Time elapsed: 2 r Zoom:
Reset

Name: Material

Material type: Diffuse

Scale 123.1925

Zoom: 100%

```
def eval(vec3 pos) vec3 :

vec3(
    noise01(
        texi() * paramScale()
    )
)
```

Skills can get you that.

Try now to multiply the *vec3()* term (the whole shader) with a new real param named **Gain**, ranging from zero to one. This exercise is optional.

2. Basically, a function is an autonomous portion of code that can be invoked in other portions in order to provide them with a certain data through a predefined process, without having to rewrite the whole thing again and again each time that this processed data is needed.

Welcome into a new dimension

Strokes across a surface. Even though jittered, you probably had a better idea of a noise function in shading.

Indeed, it was convenient to call *texi()* or *texj()* "rulers" because each is one-dimensional. Together, they are the constituents of the UV entity[3].

ISL refers to the first UV set of a geometry as *tex()*[4].

Now, our notion of a ruler is becoming more abstract. Let's stick to the naming conventions and call it *UV coordinates,* or *UV set* alike.

image: scalable 2d noise at once.

3. Such a composite entity is sometimes referred to as *construct*.
4. *tex()* is an alias for *getTexCoords(0).* The integer between the parenthesis designate the number of the UV set of choice for the geometry, starting from zero.

Dimension the Third