# SnaX

## Tutorial

Tutorial 1: The Basics!

Revision 4

Frank-Vegar Mortensen

## Introduction

This tutorial will teach you the very basic of SnaX to help you get started using this programming tool. We will create a very simple application with a scene containing a textured, rotating sphere and a user navigable camera. A sample screenshot can be seen in Figure 1. Please also read the user manual for further details about how SnaX works! The essentials to complete this tutorial are highlighted in yellow!

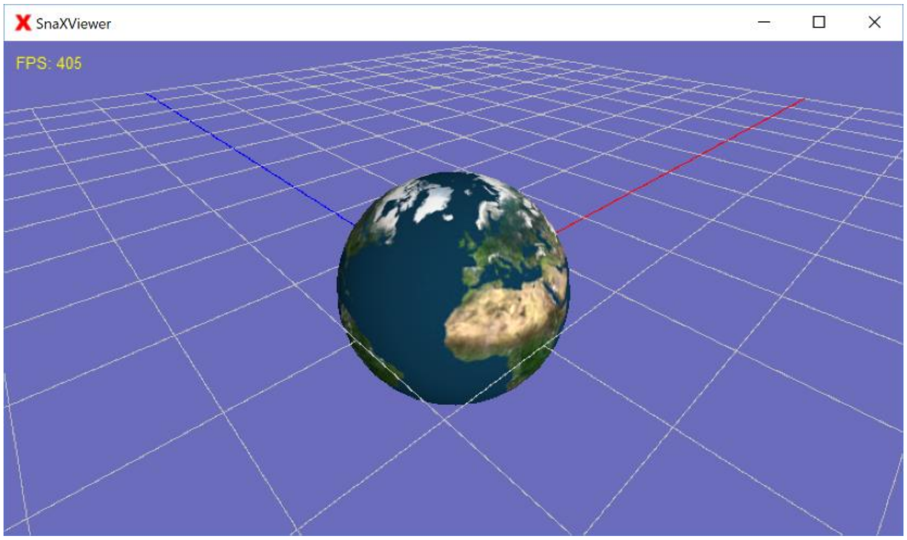


*Figure 1 The resulting application.*

## Tutorial

### Step 1

The first thing we have to do is to start up SnaX Developer. This should look something like Figure 2. The different elements of the Developer are explained in the user manual, so we should not talk too much about it here.

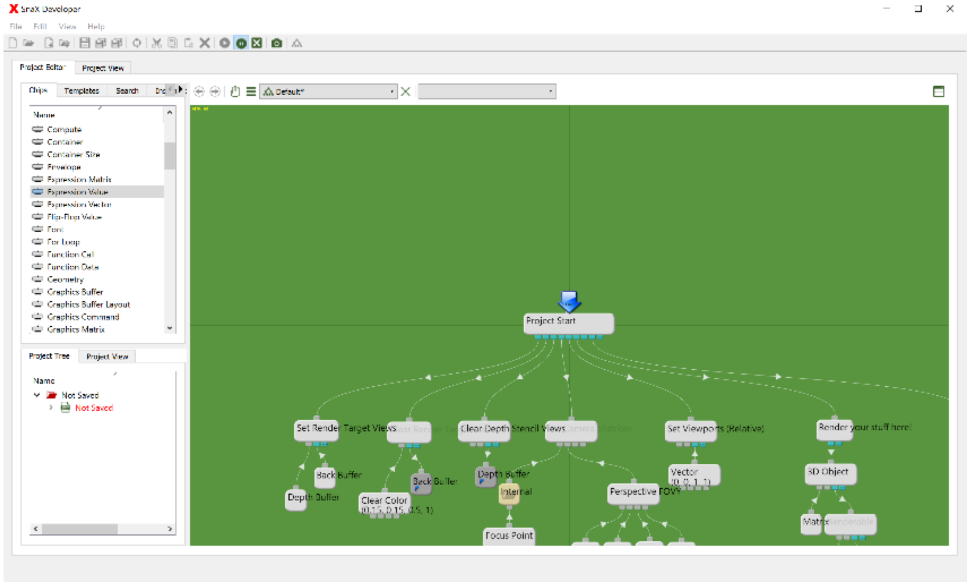


*Figure 2 The SnaX Developer.*

## Step 2

We are now ready to start programming our first application! SnaX is all about adding, configuring and connecting different type of *chips* in workspaces called *classes*. The green area, the *editor view*, in the Developer is at startup a default class with a simple program to render the famous *Utah teapot*! Try to open one of the two project views to see what the program does. If everything is correct and you open the small project view in the lower left corner of the Developer, you should get something looking like Figure 3. The benefit of the smaller view is that you can keep working in the editor view at the same time as your project keeps running. When one of the two project views are open, the *start chip* of the project is being *called* once every time the view is to be redrawn. This could happen just a few times a second to many hundred times, depending on the complexity of your program. Different type of chips does different thing when being called, but in this case the start chip (A `Caller` chip) does nothing but call its *children* in the order they are connected. Chips and connections that are currently active when the project runs, are highlighted.

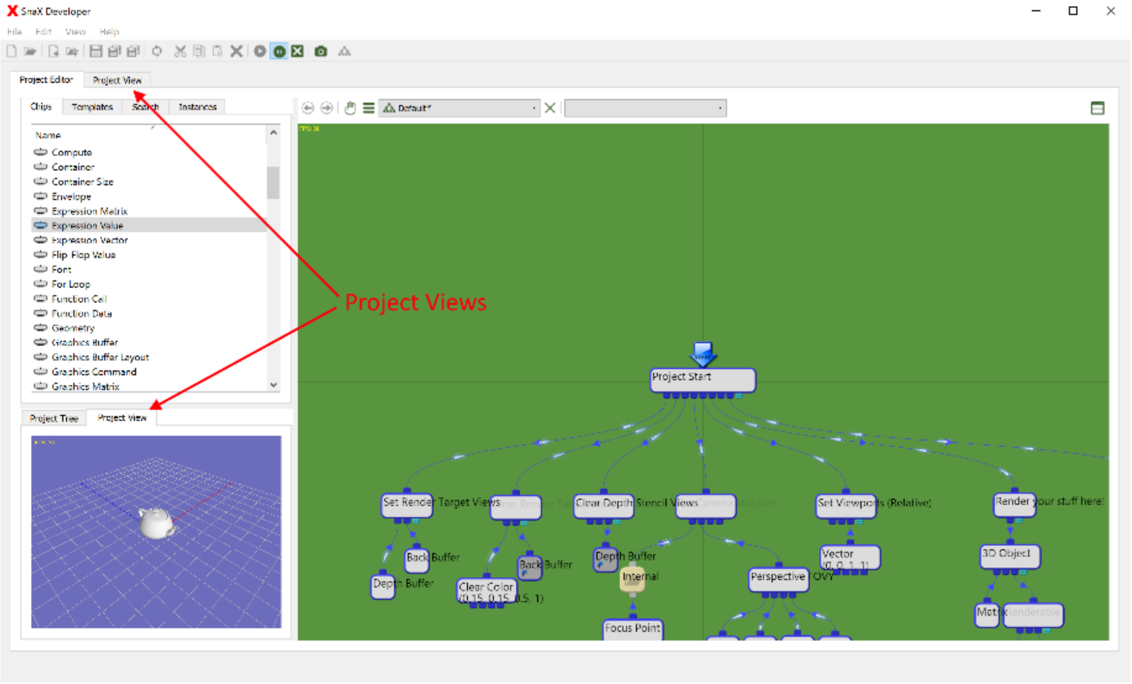


*Figure 3 The default program running in the project view.*

Let’s have a quick look at the different elements of this program. Looking at the editor view we see lots of chips linked together by curved lines called *connections*. These allow the different chips to communicate.  Please see the user manual for information about how to navigate the editor view!

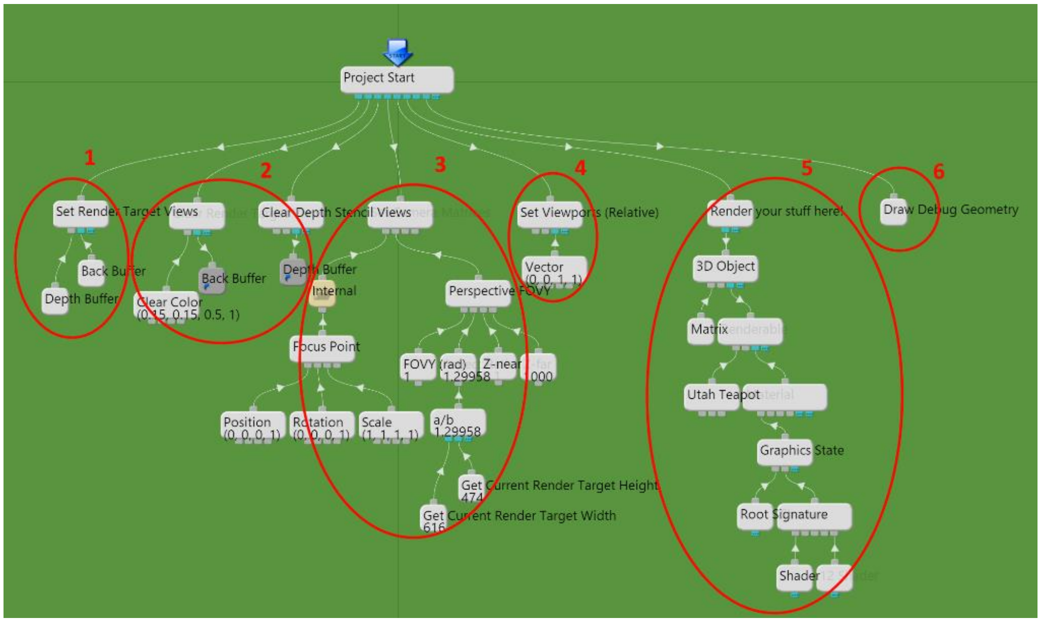


*Figure 4 The different parts of the default program.*

If we have a look at Figure 4, we can try to understand what the different parts of the program does:

1. Sets the *back buffer* (The window we render to) and a compatible *depth stencil texture* as the current *render target*.
2. Clears the back buffer with a bluish color and the depth stencil view texture with a default clear value.
3. Sets up a *camera* with its *view* and *projection matrices*. The view matrix defines the position and orientation of the camera, and the projection matrix defines the lens. The camera is movable using the mouse. The logic for this is hidden in the 'Internal' folder. Try have a look inside and see if you can understand what is going on!
4. Defines a *viewport* to cover the entire size of the render window.
5. This is the stuff that actually renders the teapot. It contains elements like the *world matrix* (position and orientation of the teapot), *geometry* (vertices and triangles), *material*, *graphics state* and *shaders*. *Textures* belong here as well, but for the teapot we have none.
6. Render debug geometry including the default world grid.

## Step 3

The first thing we are going to do is to make the teapot spin! To do this we have to modify the *world matrix* every frame with a new orientation for the teapot. We do this by replacing the default `Matrix` chip (which by default is an *identity matrix*) with a construction made up by a `Motion`, `Vector` and an `Expression Value`! Have a look in the list of chips on the left part of the Developer - Here you will find all chips that are available for use in your application. Now, select one of each type and drag them into the editor view as illustrated in Figure 5. Then, connect them with the `3D Object` using your mouse, releasing the old `Matrix` as illustrated.

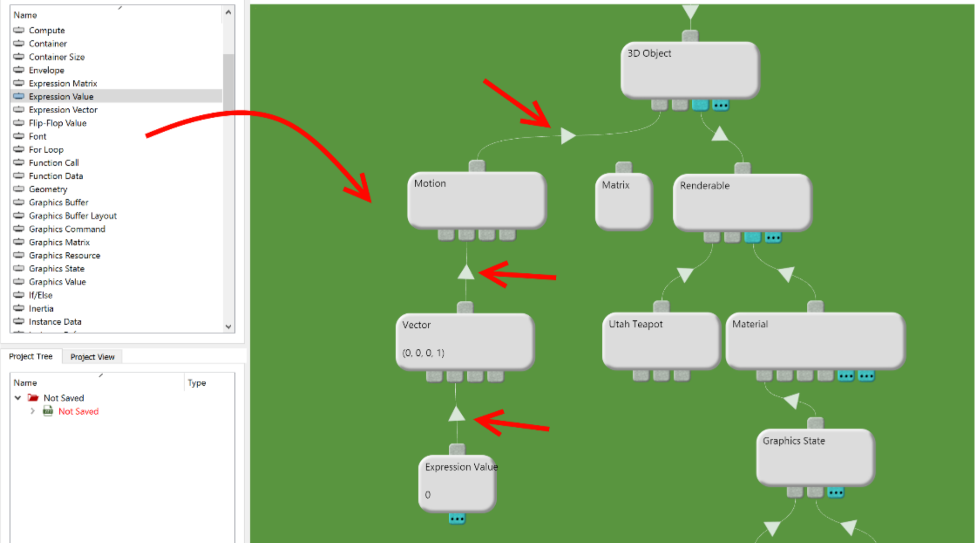
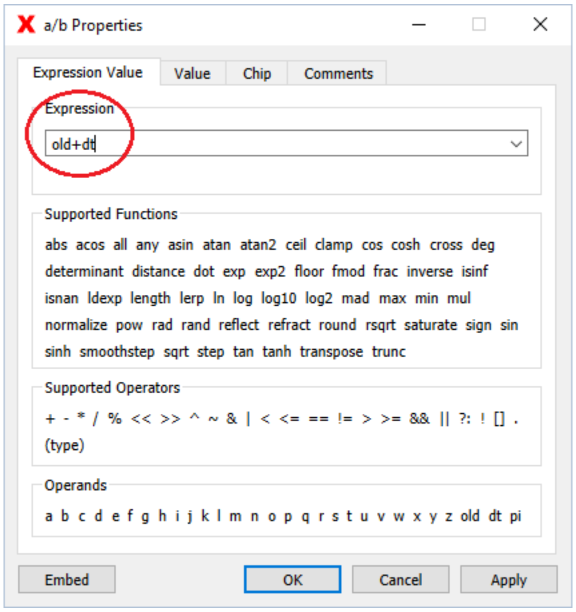


*Figure 5 Replacing the Matrix by a Motion, Vector and Expression Value.*

The `Motion` is a *specialized* `Matrix`, made up by a translation, rotation and scaling component provided by its *child connections*. Our `Vector` is connected to the rotation component, and will provide the rotation as *Euler angles*, that is, the rotation around the X-, Y- and Z-axes. When queried, the `Vector` will ask its children for values for its X-, Y-, Z- and W-components. We want to rotate around the Y-axis (Up), so we connect the `Expression Value` to the second child connection of our `Vector`.



Expression

old+dt

Supported Functions

abs acos all any asin atan atan2 ceil clamp cos cosh cross deg determinant distance dot exp exp2 floor fmod frac inverse isinf isnan ldexp length lerp ln log log10 log2 mad max min mul normalize pow rad rand reflect refract round rsqrt saturate sign sin sinh smoothstep sqrt step tan tanh transpose trunc

Supported Operators

+ - * / % << >> ^ ~ & | < <= == != > >= && || ?: ! [] . (type)

Operands

a b c d e f g h i j k l m n o p q r s t u v w x y z old dt pi

*Figure 6 The property dialog for the Expression Value.*

An `Expression Value` accepts a mathematical formula to calculate its *value*, which is single, real number, stored as a 64-bits *floating-point value (double)*. The formula is provided in the `Expression Value`'s *property dialog* that is opened by double clicking the chip itself. Enter "*old+dt*" in the expression field of the dialog, as shown in Figure 6. This will take the current value (*old*) of the chip, and add it to the time it took to render the previous frame (*dt*). This will create an angle that is increasing one *radian* per second. Try to open the Project View again. If you did everything right, you should see a spinning teapot! How about that!

## Step 4

Now, we don't want a spinning teapot, so we have to replace the teapot geometry with a *sphere*. Luckily, this is very simple to do! The geometry for the teapot is provided by the `Primitive` chip (currently named 'Utah Teapot'), which contains a large list of different geometric primitives, including the sphere. Open its property dialog, and select "Sphere" from this list. Looking at the project view again, you should be able to see the new geometry. By default, the number of vertices and triangles making up the sphere is a little low, so the sphere does not look very good. We fix this by linking a `Vector` to the first child connection of the `Primitive`. This controls the *subdivision*, i.e. the number of vertices the sphere is made up of. Open the `Vector` 's property dialog and set a number around 30 for the X and Y components. Now the sphere should look pretty smooth.

**Exercise**: Have a look at the `Graphics State` chip connected to the `Material`. Try and see if you can enable the *wireframe Fill Mode*. Play a little with the subdivision settings again, and see how it affects the geometry for the sphere. It should look something like illustrated in Figure 7.

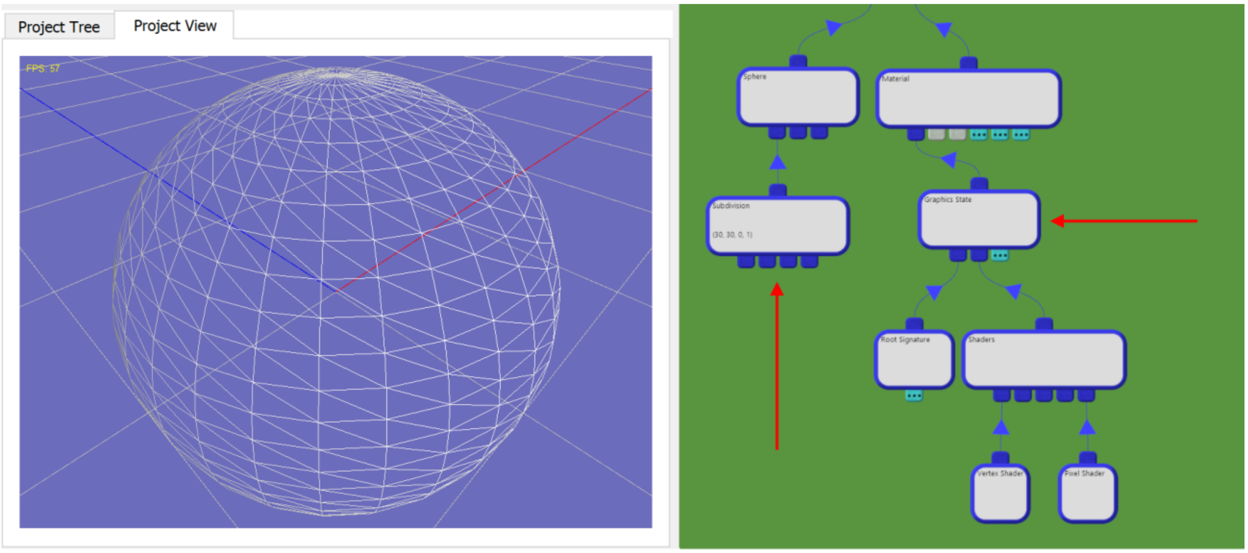
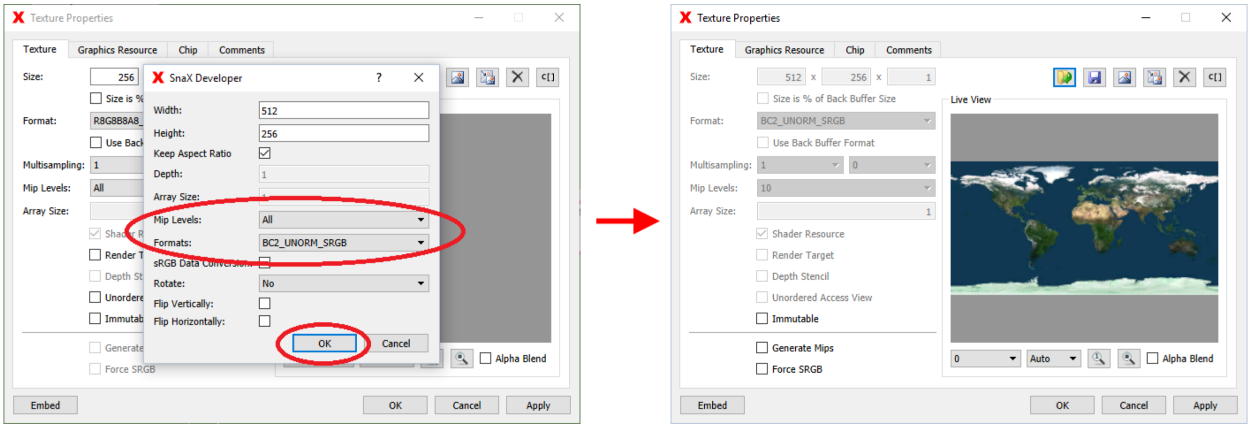


*Figure 7 Enabling wireframe mode from the Graphics State chip.*

## Step 5

Now we are going to add a *texture* to our sphere! The material system of SnaX is very generic! Nothing is predefined, so you will have to create everything yourself, including material properties, lighting, shading etc. This reflects the concept of Direct3D 10-12 where there is no *fixed pipeline* with predefined material and lighting properties as is was in older Direct3D versions. This is a good thing as is gives you complete control in how you want to build your shading system. The downside is that it requires a bit more work to get started. For a real-life project, you will reuse lots of the material system you create, so it will be easier as you get things up running. A *material* is basically a group of resources (textures, samplers, buffers etc), paired with a *pipeline state* defining how to use the resources to render the geometry.



Texture Properties
Size: 256
☐ Size is %
Format: R8G8B8A8_
☐ Use Back
Multisampling: 1
Mip Levels: All
Array Size:
☑ Shader R
☐ Render T
☐ Depth St
☐ Unordere
☐ Immutab
☐ Generate
☐ Force SRGB

SnaX Developer
Width: 512
Height: 256
Keep Aspect Ratio ☑
Depth: 1
Array Size: 1
Mip Levels: All
Formats: BC2_UNORM_SRGB
sRGB Data Conversion: ☐
Rotate: No
Flip Vertically: ☐
Flip Horizontally: ☐

Texture Properties
Size: 512 x 256 x 1
☐ Size is % of Back Buffer Size
Format: BC2_UNORM_SRGB
☐ Use Back Buffer Format
Multisampling: 1 0
Mip Levels: 10
Array Size: 1
☑ Shader Resource
☐ Render Target
☐ Depth Stencil
☐ Unordered Access View
☐ Immutable
☐ Generate Mips
☐ Force SRGB
Live View
0 Auto ☐ Alpha Blend

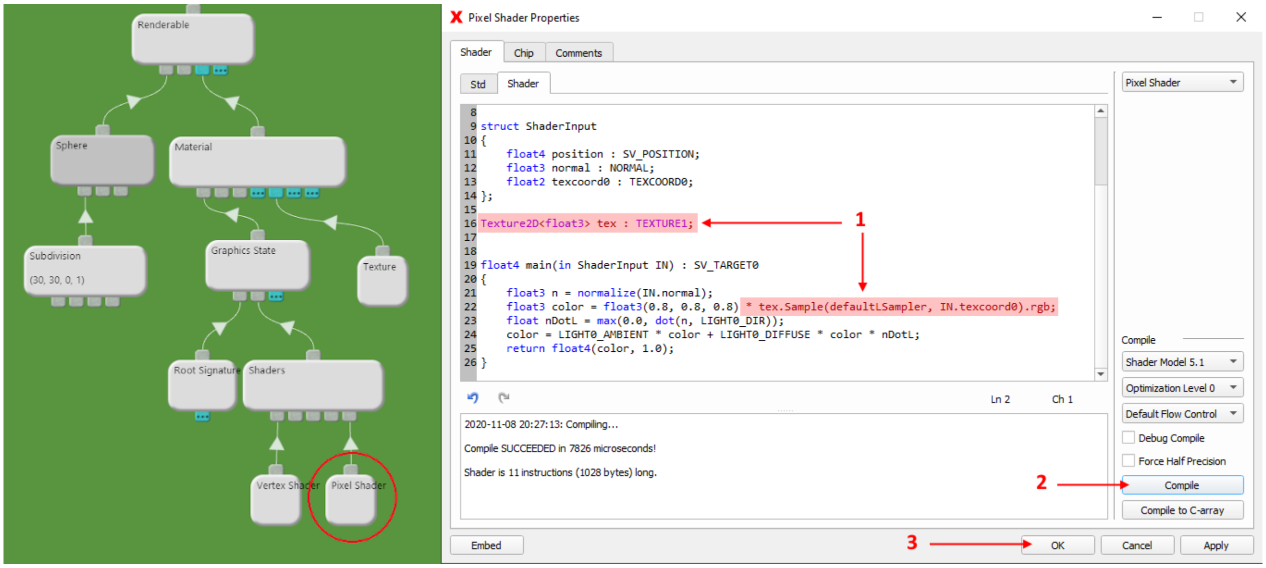
*Figure 8 Selecting format and mip level for the imported texture.*

To add a texture to our material, we have to go back to the list of chips, find the `Texture` chip, and drag in into the editor view somewhere below the `Material` chip. Connect it to the `Material`'s "*Shader Resource Views*" connector, careful not to confuse it with the connector for the "Constant Buffer View". Open the `Texture`'s property dialog, and click the 📂 (Load Image From File) button. Select the 'earth.jpg' file located in the Tutorial folder in SnaX's install directory. Any other jpg-file will work as well of course! By default, SnaX uses a *sRGB* back buffer (Google it!). This means that the textures we use should also be of a sRGB format. You should therefore change the format of the imported image from R8G8B8A8_UNORM to R8G8B8A8_UNORM_**SRGB**, or even BC2_UNORM_**SRGB** to apply texture *compression* as shown in Figure 8! These cryptic terms are reflecting the texture formats defined in the Direct3D API and should therefore be familiar to people with a background in graphics and Direct3D. Also, make sure "*Mip Levels*" is set to "*All*" to the best rendering quality.

Press OK. You should now see the imported texture in the "Live View" widget of the dialog.

## Step 6

We've added the texture, but we still don't see it, right? No reason to panic, we need to update our *shaders* as well! Shaders are programs running on the *GPU* (your graphics card) and are responsible for doing the actual rendering. They define which textures to use, and how to apply them to the geometry you render. The current shaders do not evaluate our texture, so we need to modify the *pixel shader* slightly to do this. Editing shaders are the only place in SnaX where you will need to do some traditional programming, at least for now – Visual shader programming are work-in-progress!



```
struct ShaderInput
{
    float4 position : SV_POSITION;
    float3 normal : NORMAL;
    float2 texcoord0 : TEXCOORD0;
};

Texture2D<float3> tex : TEXTURE1;


float4 main(in ShaderInput IN) : SV_TARGET0
{
    float3 n = normalize(IN.normal);
    float3 color = float3(0.8, 0.8, 0.8) * tex.Sample(defaultLSampler, IN.texcoord0).rgb;
    float nDotL = max(0.0, dot(n, LIGHT0_DIR));
    color = LIGHT0_AMBIENT * color + LIGHT0_DIFFUSE * color * nDotL;
    return float4(color, 1.0);
}
```

```
2020-11-08 20:27:13: Compiling...

Compile SUCCEEDED in 7826 microseconds!

Shader is 11 instructions (1028 bytes) long.
```

*Figure 9 Modifying the pixel shader to evaluate our texture.*

Open the Pixel Shader's property dialog and add the code as highlighted in Figure 9:

- `Texture2D<float3> tex : TEXTURE1;`
- `float3 color = float3(0.8, 0.8, 0.8) * tex.Sample(defaultLSampler, IN.texcoord0).rgb;`

`TEXTURE1` and `defaultLSampler` are defined in the *Std-page* of the shader dialog. This page contains macros and default objects to be included in the shader code. `TEXTURE1` refers to the first texture (Shader Resource View) connected to the material, and `defaultLSampler` defines a default sampler with linear filtering and clamping of texture coordinates.

Press 'Compile'. If there are no errors, the "Compile SUCCEEDED" message is displayed. Click OK to exit the dialog.

Voila! You should now see the spinning globe if you open the project view!

## Step 7

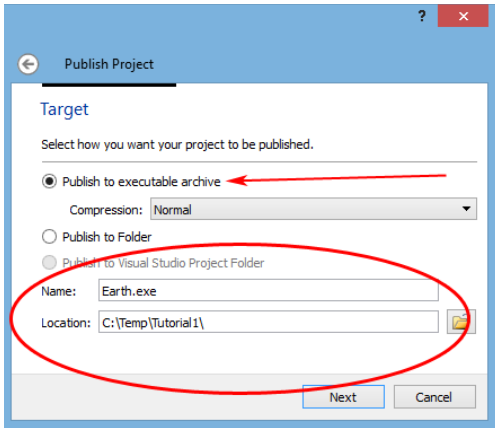
The last step in this tutorial is to *publish* the project to an *executable file*.

The first thing we have to do is to save the document containing our class to a file. Make sure the editor view (green area) has focus, then select 'Save' from the toolbar or the File-menu to bring up the file dialog. Select a file name and location. Preferably, save the file to an empty folder. You can call the file 'Earth.m3x'. The *m3x* extension means that this is an XML-based project file. You can open it in any text editor, but be very careful if you try to modify anything manually – It is very easy to render the file unreadable by SnaX.

To publish the project, we have to select 'Publish…' from the File-menu. This will bring up the *publish wizard* to configure the publishing process. This is quite easy, and for the most part you just step through the process by clicking 'Next' on most pages. The wizard is described in detail in the user manual.

The most interesting step is Step 3. This is where you select the *format* of the published project. We will go with the default, which is a *self-extracting executable archive*. When the resulting exe-file is launched, the project is extracted to a temporary directory and the bundled *SnaXViewer* is launched with your project as input.

Verify the target file name and location as indicated in Figure 10 before proceeding. The last step is about selecting which project files to include with the publication. We only have one file, and it should be selected by default, so go ahead and click 'Next' right away. Now, finish the publish wizard and let it do its job for a few seconds. You should be met by a nice message box congratulating you with a successful build!



Target

Select how you want your project to be published.

● Publish to executable archive

Compression: Normal

○ Publish to Folder

○ Publish to Visual Studio Project Folder

Name: Earth.exe

Location: C:\Temp\Tutorial1\

*Figure 10 Selecting format and target location as step number three.*

Now, use your file browser to find the new exe-file you just created. It's located in the folder you selected in step 3 of the wizard. Start the file, watch the project extract and launch! Your application should look something like Figure 1.

This is the end of this tutorial. Try looking at the project again and see if you understand how everything works. Try tweaking and changing parts of the project and see what effect it has. That's the best way to learn how to program SnaX!

Enjoy!