Shaders - Beginner

Introduction
Many games often use shaders to create in game effects. More than likely you will have heard of this term being used. In this tutorial we will examine what shaders are and see how they can be utilised from within Dark GDK.

What is a shader
First of all let's being with some definitions - a shader is a file that tells Dark GDK how to draw an object. For example a simple shader may state that every pixel on the object should be drawn bright red, whilst it may not look very useful it does demonstrate that you now have complete control over the color of every pixel used to draw the object.

More specifically a shader contains two main parts, a vertex shader and a pixel shader. The vertex shader is responsible for positioning the object and its vertices in the world, without which commands such as dbPositionObject and dbRotateObject would have no effect, and produces a small set of values which are passed to the pixel shader. The pixel shader then takes these values, along with any textures applied to the object, and paints the object onto the screen in pixels.

The pixel shader part is often the part that is the most complex as, for example, it can also be responsible for applying light to an object.

Every pixel that is covered by the object is calculated within the pixel shader applied to that object. When an object does not use a shader the pixel color is calculated by the "fixed function pipeline."

Why use shaders
Before shaders became popular, rendering was done using the 'fixed function pipeline' which could be described as a shader hardcoded into the graphics card. It contained a fixed set of instructions that lit and textured your objects based on a few parameters passed to it such as light color and textures. However this fixed function could not be changed without changing the hardware and was based upon what was thought everyone wanted their objects to look like.

However, programmable shaders allow you to bypass the fixed function pipeline and supply your own functions for how you want your objects to look. What makes shaders useful is their flexibility, you can color the pixels of an object using any function that will fit inside a pixel shader, from a solid color to

a lightmapped reflection shader; and the size of the pixel shader instruction limit is increasing with every new pixel shader version.

In the future the fixed function pipeline will no longer exist, as is the case with DirectX 10, since shaders are now far more useful than the fixed function pipeline was, the hardware space once used for the fixed function can now be used to help increase the speed and size of user defined shaders. Also a shader can be written that could replicate any feature the fixed function pipeline currently has.

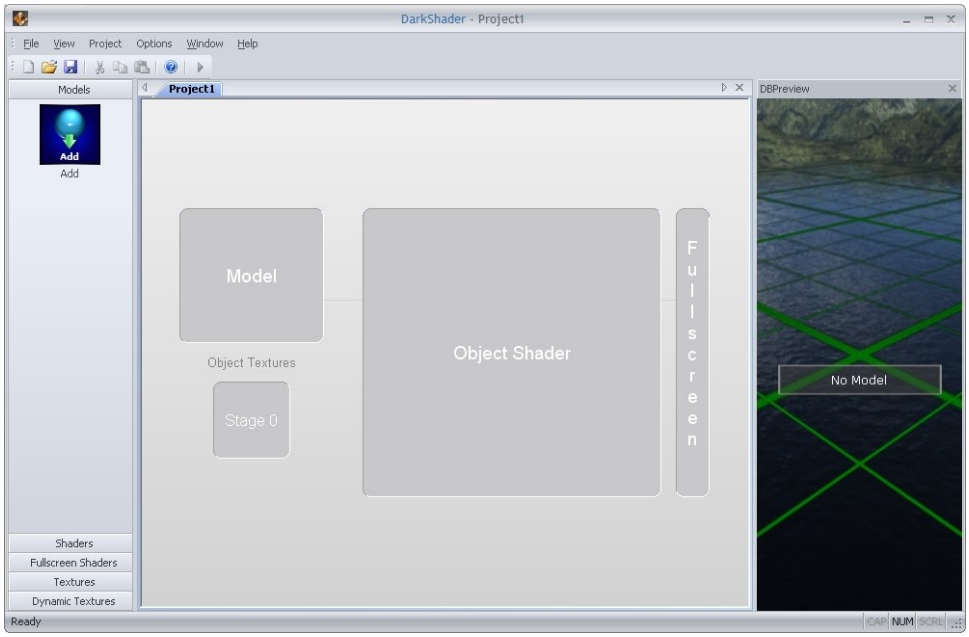
So a shader takes control of the positioning, lighting and texturing of an object that would normally be handled by fixed processes and lets you specify your own method of texturing and lighting you want applied to your objects.

Creating shaders
There are numerous options available for creating shaders. You can create them manually, find an existing shader on a game development website and modify it for your own purposes or use a visual editor. For our tutorial we're going to use a visual editor. While we won't go into a great amount of technical detail regarding exactly what the shader is doing, this tutorial will demonstrate the process of using shaders in Dark GDK.

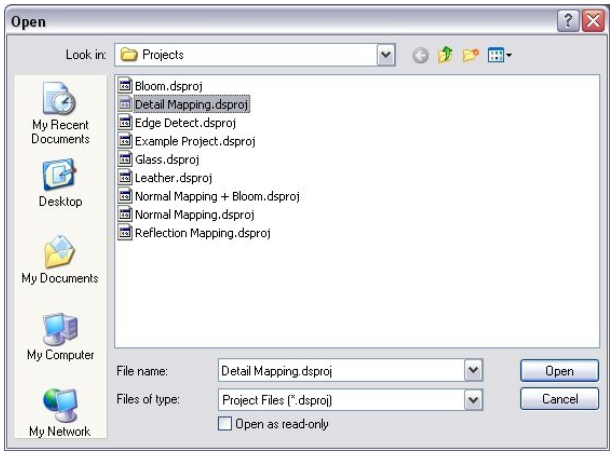
The tool we are going to use to create a shader is called Dark Shader. This is a product that has been developed by The Game Creators. Its aim is to allow end users to visually create, modify and export shaders. By using this tool you don't necessarily need to know all the specifics of exactly what is going on but you can still get the end result. It is a good starting point for those individuals who are new to shaders. For more information on Dark Shader visit http://darkbasicpro.thegamecreators.com/?f=darkshader

Getting Started
Dark Shader is going to be used to create a shader that we can work with in our project. The shader we create will place two textures onto an object and allow them to be blended together. In the next few stages we'll see how a shader can be created and exported ready for use.

After Dark Shader has been launched you will get to see the following:

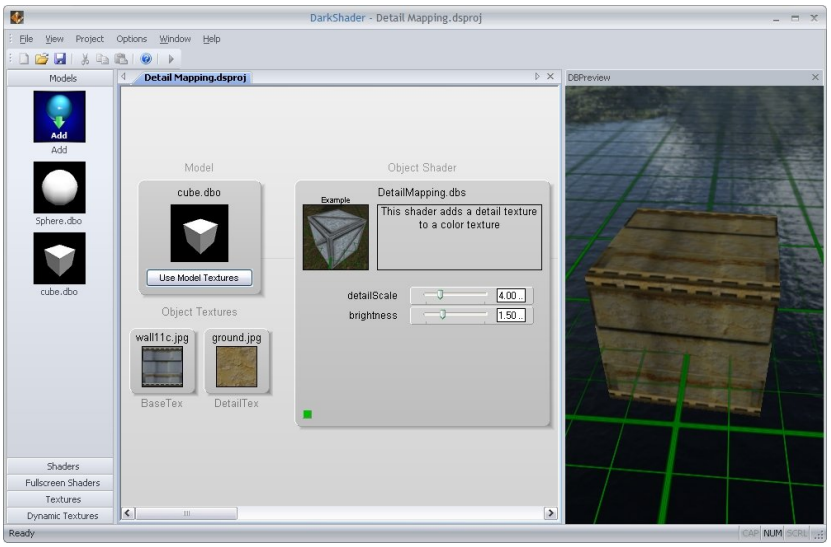


From this point onwards you can build shaders and experiment with all kinds of settings. However, for our purposes we're going to use Dark Shader in a very simple way. We will load one of the existing projects and select the Detail Mapping shader:



This shader will apply two textures to an object in the scene. When the textures are applied together they will be blended to produce the end result.

Here is the initial set up for the detail mapping shader:



The project has been set up with a cube, it has been given two textures and then the detail shader has been applied. The detail shader takes the two textures and maps them on top of each other and blends them together to create a final image.

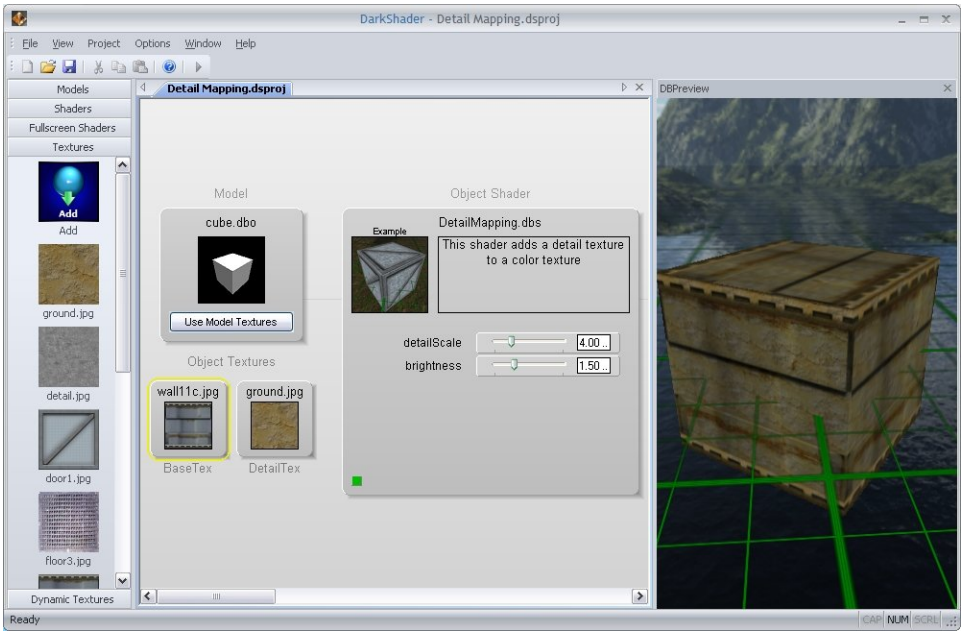
To help understand this lets take a closer look at what happens with our images:

In the first stage we have our two separate images.

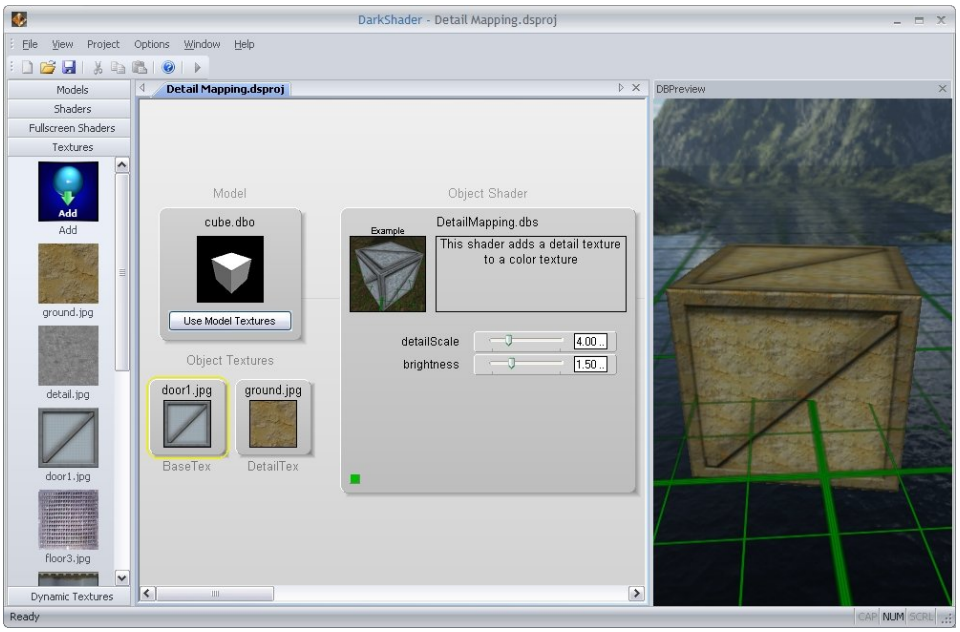
In the second stage we see that they are combined together and place on top of each other with the base texture being on the bottom and the detail texture

being on the top. In the third stage we see the results of combining these textures together. It is a very simple technique but one that is useful for many different circumstances.

To get a clearer view of this effect we will change these textures. To do this we need to click on the texture and then we can select one of the supplied textures or import one of our own:

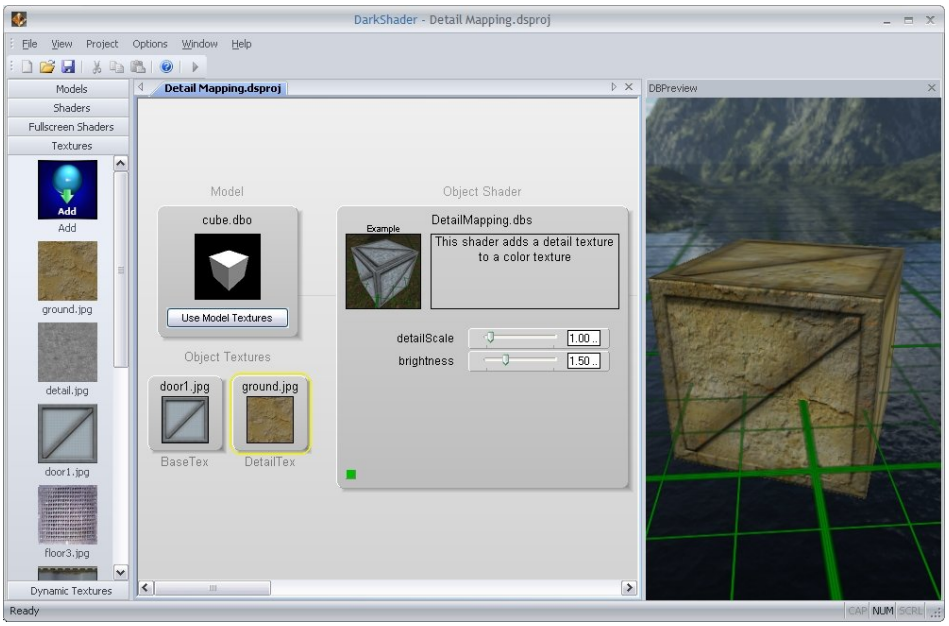


For our example we will change the wall texture to a door texture:

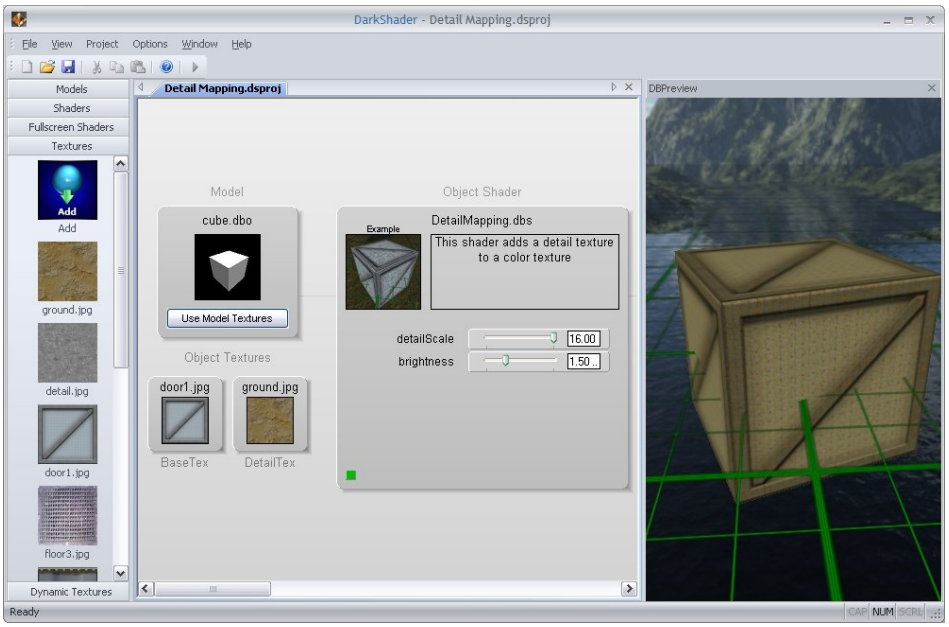


Now we have two very distinct images the effect should become much clearer. You can get a much clearer view of how these two textures are being merged together to create the final result.

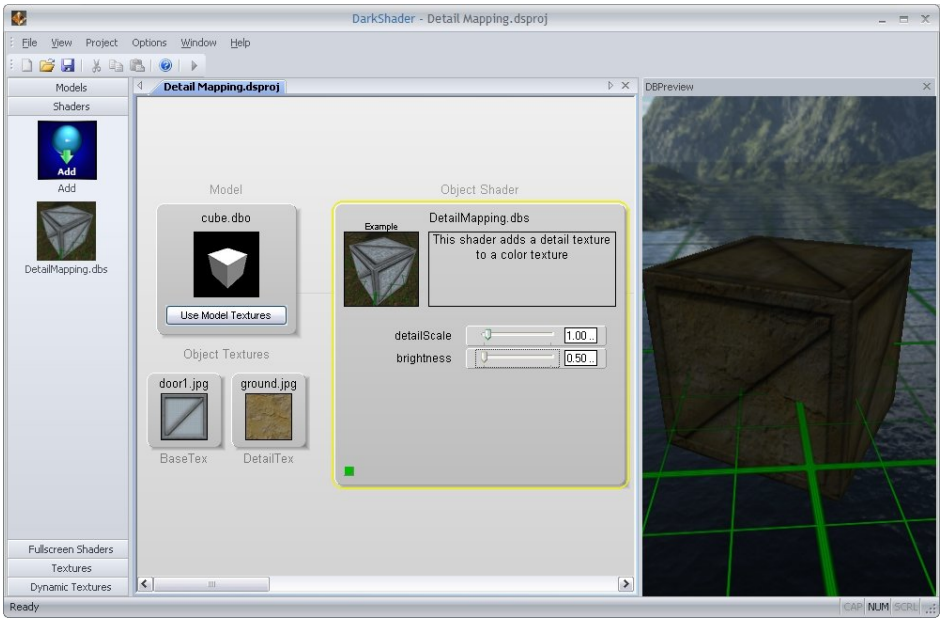
We can see in the object shader section that this shader contains two adjustable properties. There is a property named detailScale and one named brightness. The detailScale property controls how our second texture is scaled. The initial value for this property is 4 which results in it being tiled over each face of the cube by 4 x 4. To see this effect more closely the detailScale can be adjusted to a value of 1:



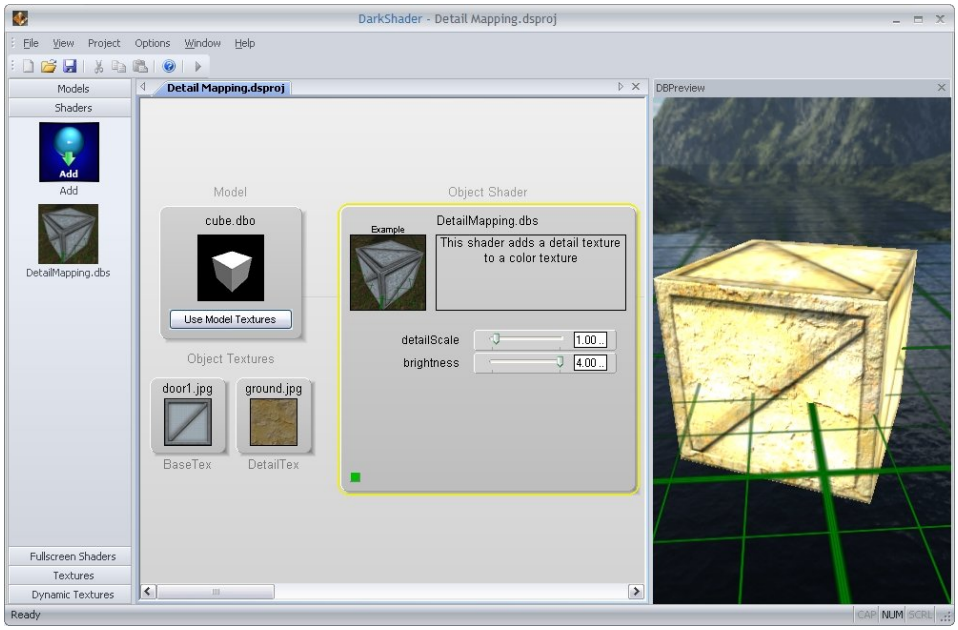
Notice how this time the second texture has been stretched over each face of the cube and we no longer get it repeating. A quick change to this value from 4 to 16 results in the second texture being repeated by 16 x 16 over each face of the cube. Notice how you can see the small repeats on the cube:



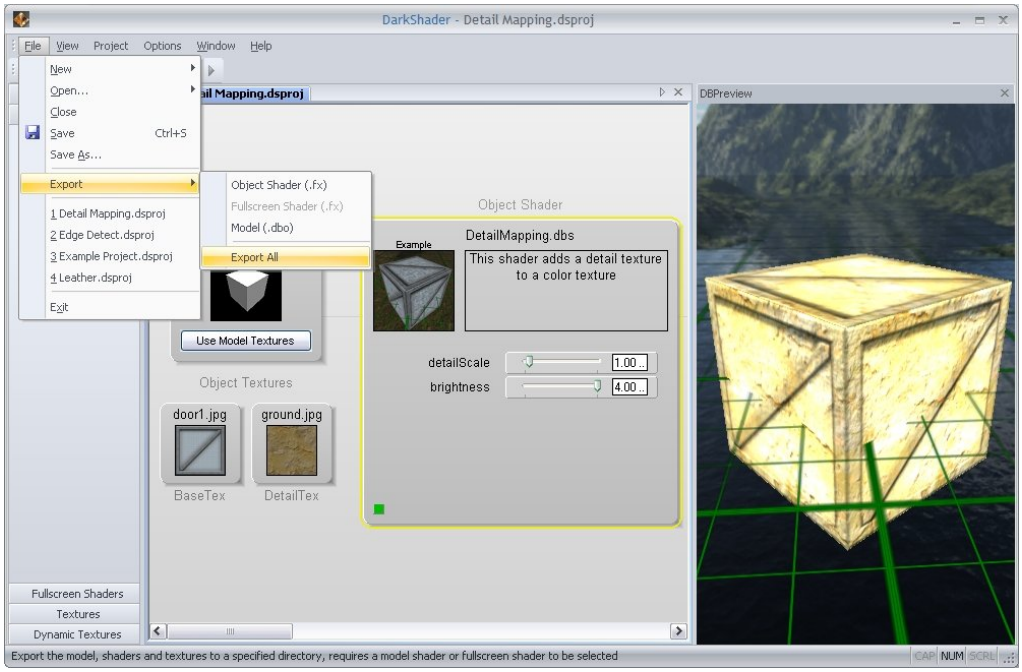
The second property we can change allows us to set the brightness of the textures. Let's change our detailScale back to 1 and adjust our brightness level to 0:



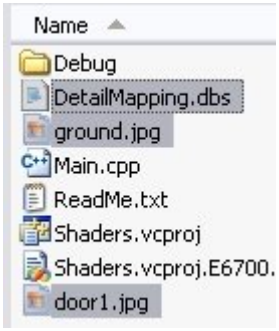
By doing this we have made the output result very dark. By increasing this value we can make the result much brighter as in this example where the brightness is set to 4:



This shader can be exported by selecting the File menu, going to Export and selecting the Export All option:



Once the files have been exported we can copy them across to our project:

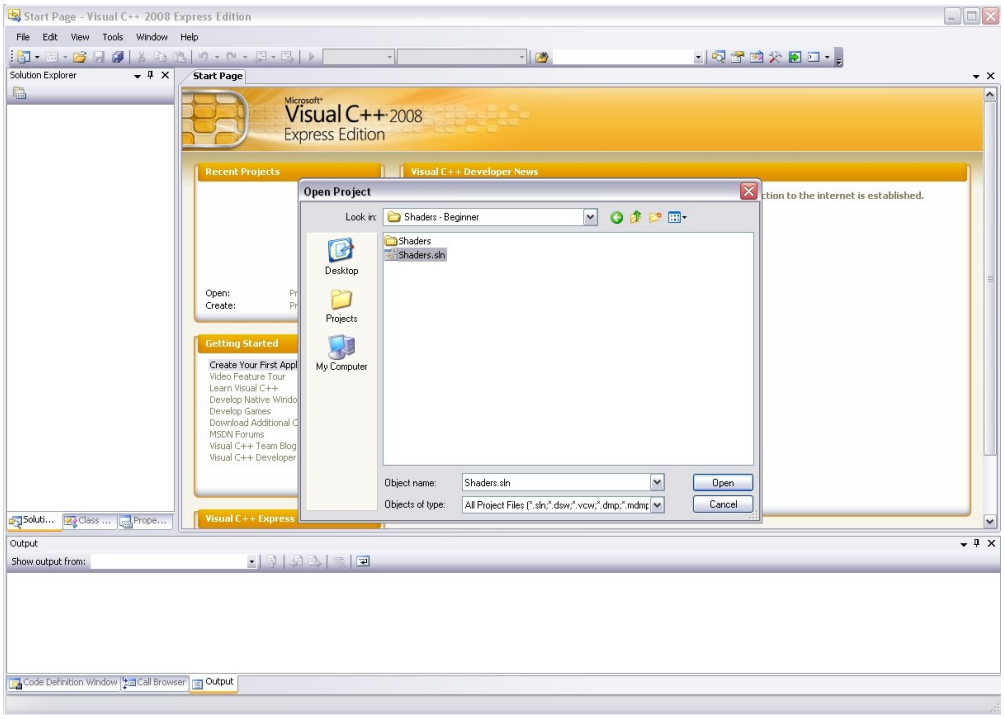


The main file exported is named DetailMapping.dbs. This file constructions instructions for our shader. The other two files are the textures that were applied to our object.

To conclude, we have seen a pre built detail mapping shader loaded into Dark Shader, we have seen how the effect works and examined its two main properties.

Opening the project
A project has been created with some example code so we can see how to use this shader in Dark GDK.

- Launch Microsoft Visual C++ 2008
- Go to the File menu and select Open and then Project.
- From here navigate to the directory where Dark GDK is installed. By default this is C:\Program Files\The Game Creators\Dark GDK.
- Now go into the Tutorials folder and then the Shaders – Beginners. folder.
- Finally open the solution file for Shaders.



Overview of code

The code for this program will set some initial properties, make a cube, apply the shader effect and then enter our main loop where the cube is rotated:

```
#include "DarkGDK.h"

void DarkGDK ( void )
{
	dbSyncOn ( );
	dbSyncRate ( 60 );

	dbMakeObjectCube ( 1, 1 );
	dbLoadEffect ( "DetailMapping.dbs", 1, 1 );
	dbSetObjectEffect ( 1, 1 );

	while ( LoopGDK ( ) )
	{
		dbTurnObjectLeft ( 1, 0.5 );
		dbSync ( );
	}
}
```

Loading and applying a shader

The main calls we need to concentrate on are dbLoadEffect and dbSetObjectEffect. They are called after our program has been set up and a cube object has been made.

```
	dbLoadEffect ( "DetailMapping.dbs", 1, 1 );
	dbSetObjectEffect ( 1, 1 );
```

The first function dbLoadEffect is used to take an effect file and apply it to an object. It has three parameters with the first being a filename, the second an

ID number for the shader and the final parameter determines whether any textures that the shader defines will be loaded. In our example program we made the shader using Dark Shader, exported the project and copied the files into our program directory. The effect file that was exported from Dark Shader is called “DetailMapping.dbs” so this is used as our first parameter. We need to supply our effect with an ID number for the second parameter. ID numbers for effects our separate from objects therefore we can have an object loaded into ID 1 and load our effect with an ID of 1. The final parameter is used to specify whether any textures in the shader will be loaded. Our detail mapping shader was set up to use two textures and these were exported and placed into our project directory. So for our example we rely on the effect using any associated textures therefore we use a value of 1 for this parameter. Later on when this effect is applied to an object these textures will also be applied. If we used a value of 0 any texture information in the shader would be ignored.

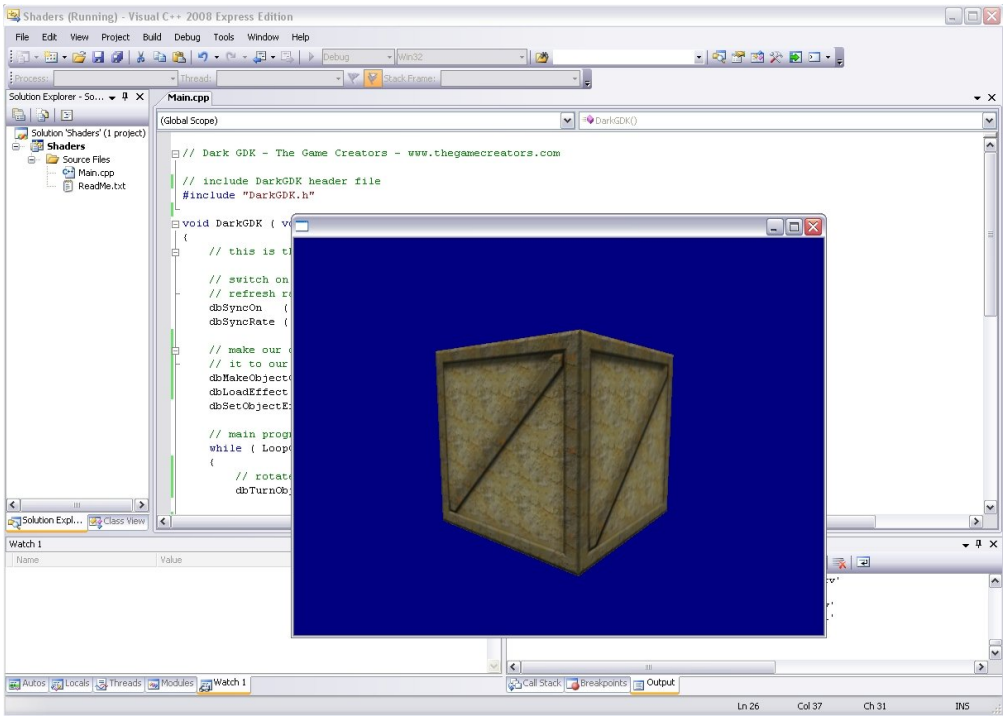
The second function is called dbSetObjectEffect. It takes two parameters. The first parameter is used to specify an object ID number while the second parameter is used to specify an effect ID number. Our cube was made using an ID of 1 and our effect was loaded with an ID of 1 so both our parameters will be 1.

Main loop
The remainder of the program is our main loop:

```
while ( LoopGDK ( ) )
{
      dbTurnObjectLeft ( 1, 0.5 );
      dbSync ( );
}
```

It is a very simple loop that consists of two function calls. The first is a call to dbTurnObjectLeft and as you can see from the source code we’re using an ID of 1 and a rotation value of 0.5. This will have the result of turning the cube on its Y axis. We could have also used dbRotateObject or dbYRotateObject to achieve this effect. The second function call is to dbSync. This will update the contents of our screen.

Running the program
With all our code in place it’s time to run the program and see how our shader is displayed in Dark GDK. Go to the Debug menu and select Start Debugging:



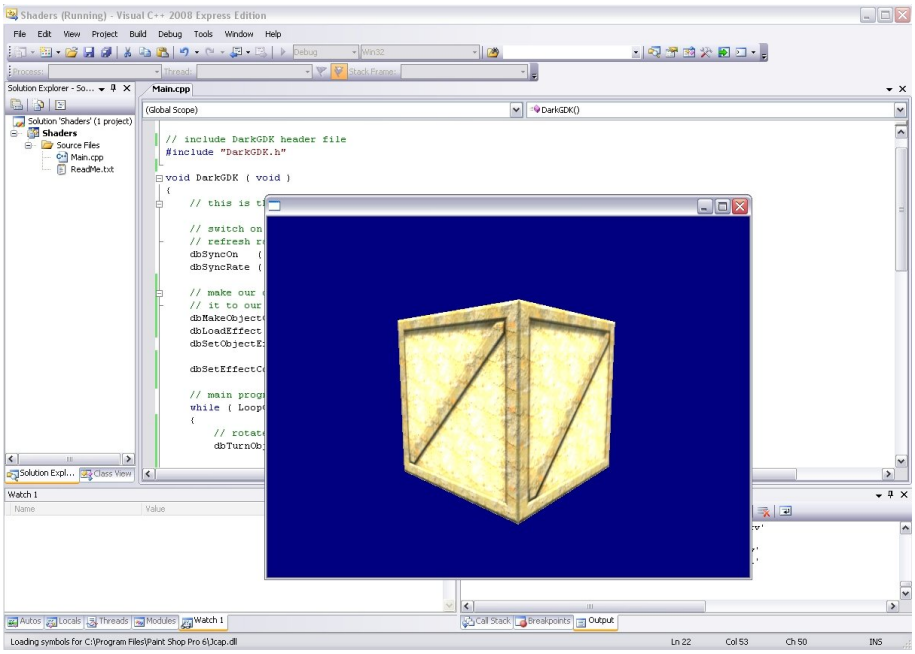
We can see our cube has been loaded and the shader effect has been applied to it resulting in two textures being blended together.

Modifying shader properties

When our shader was created it had two main properties that we could modify in Dark Shader. These two properties brightness and detailScale can be accessed from within Dark GDK by using the function dbSetEffectConstantFloat. This function takes three parameters. The first is the ID number of the effect. The second parameter is the name of the property. The third parameter is the new value to be assigned to the property. Try adding this line before the main loop:

```
dbSetEffectConstantFloat ( 1, "brightness", 4 );
```

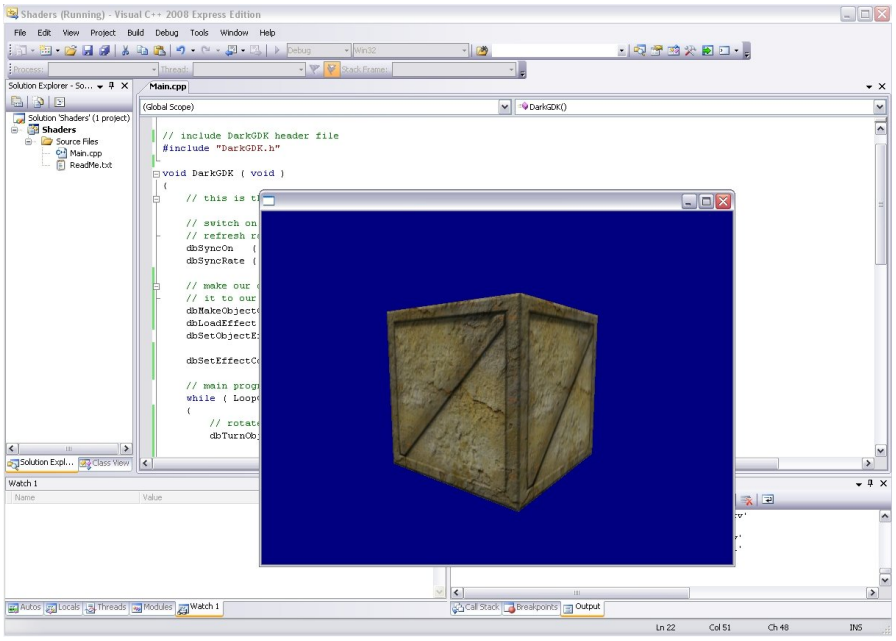
Run the program again and see how adjusting this property of the shader has affected the output result:



It is also possible for us to alter the detailScale property by using the same function. In this line of code the scale is changed to a value of 1:

```
dbSetEffectConstantFloat ( 1, "detailScale", 1 );
```

Try adding this line of code into the program before the main loop and remove any alterations to the brightness property. The new result is:



Conclusion

In this tutorial we have looked at the process of visually creating a shader, exporting it and using it within Dark GDK.