Shaders - Advanced

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

What is a shader
First of all lets 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.

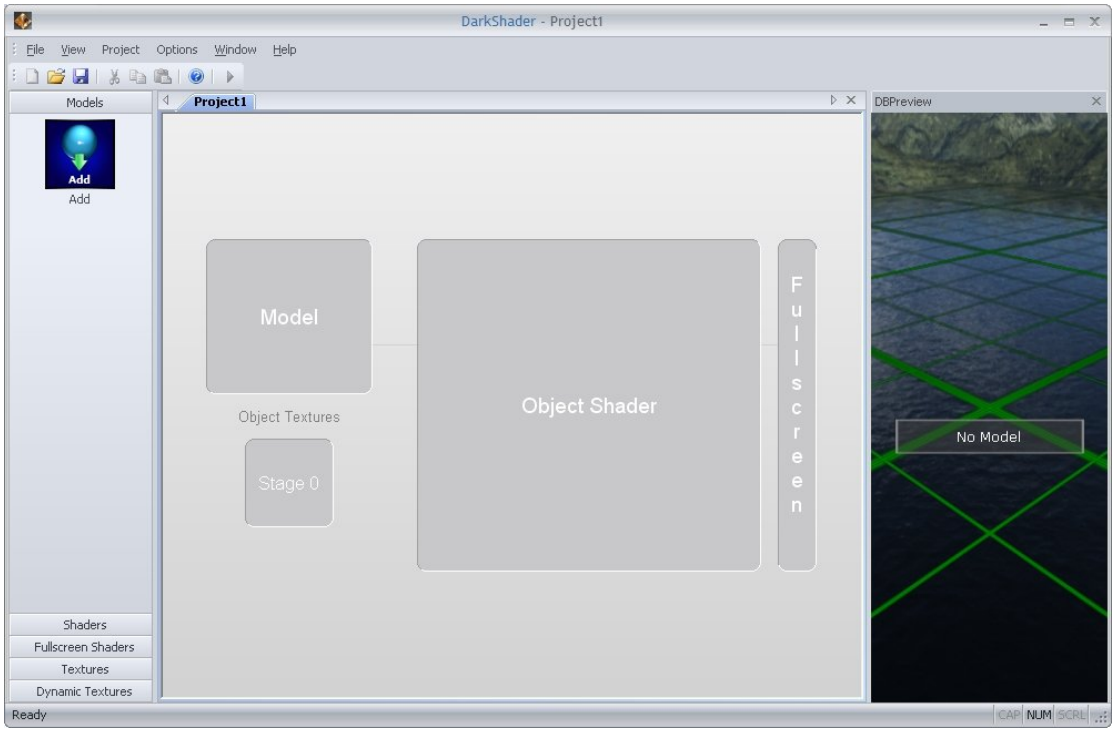
So a shader takes control of the positioning, lighting and texturing of an object that would normally be handled by fixed processes and lets you specify your own method of texturing and lighting you want applied to your objects.

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.

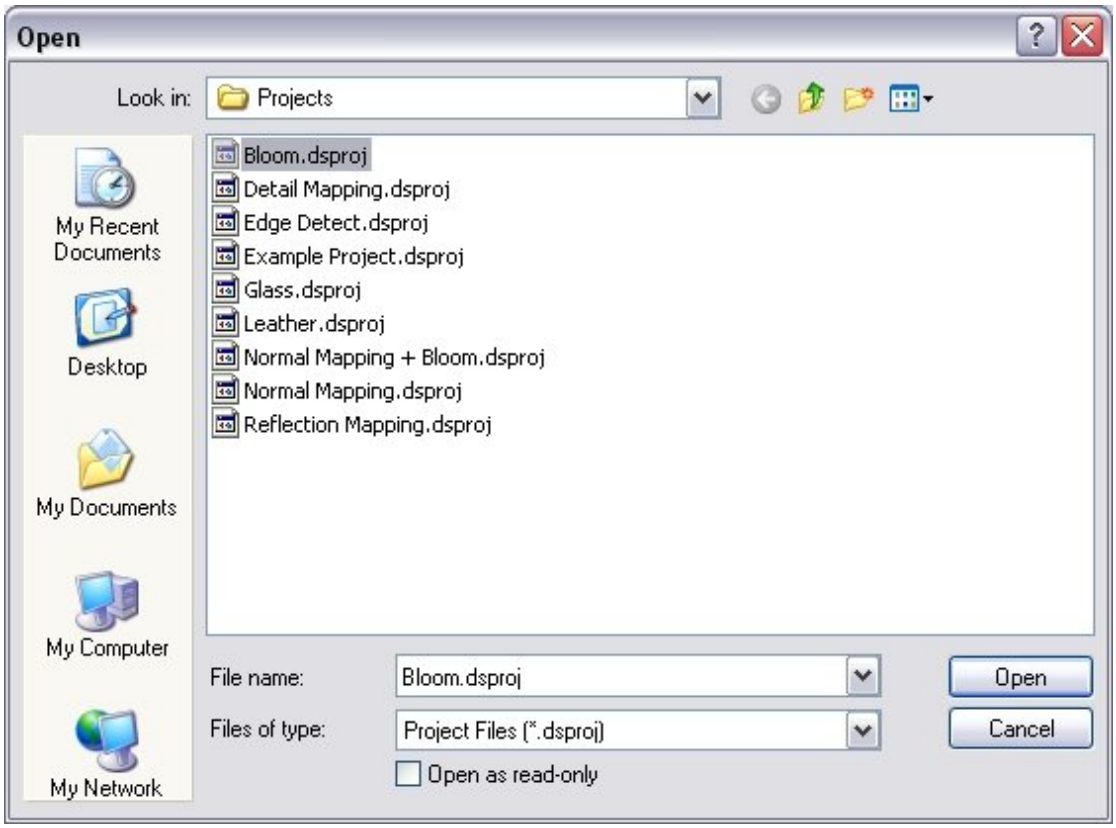
The tool we are going to use to create a shader is called Dark Shader. This is a product that has been developed by The Game Creators. Its aim is to allow end users to visually create, modify and export shaders. By using this tool you don't necessarily need to know all the specifics of exactly what is 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. In the next few stages we'll see how a shader can be created and exported ready for use.

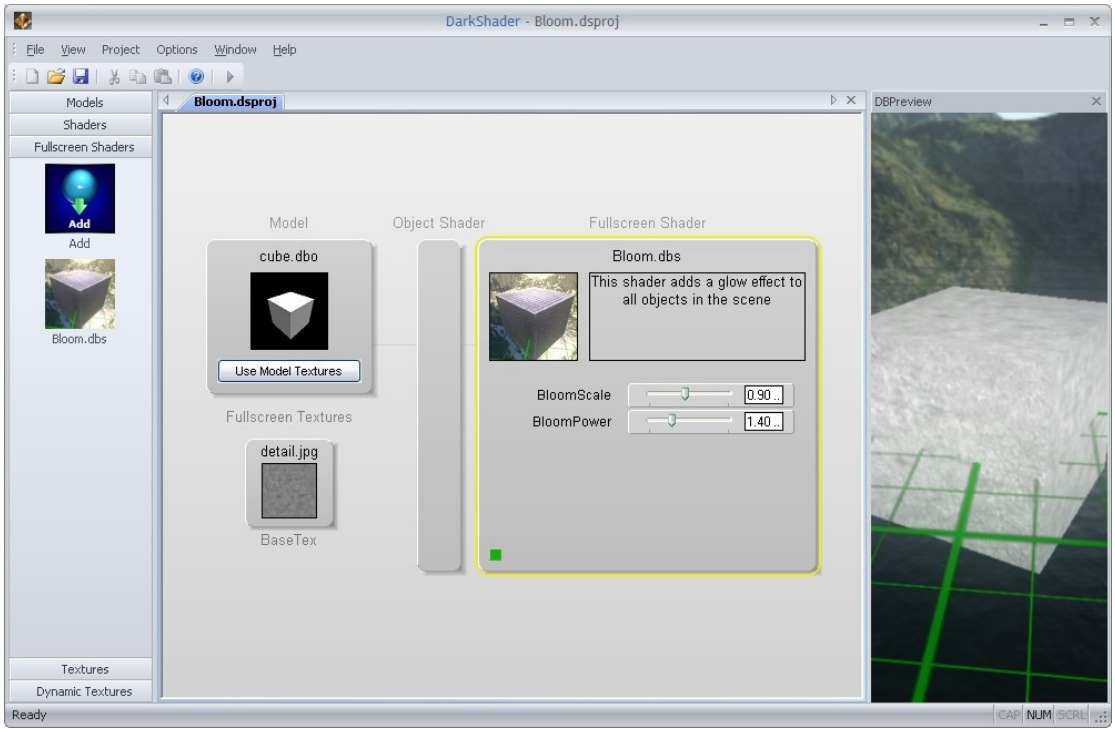
After Dark Shader has been launched you will get to see the following:



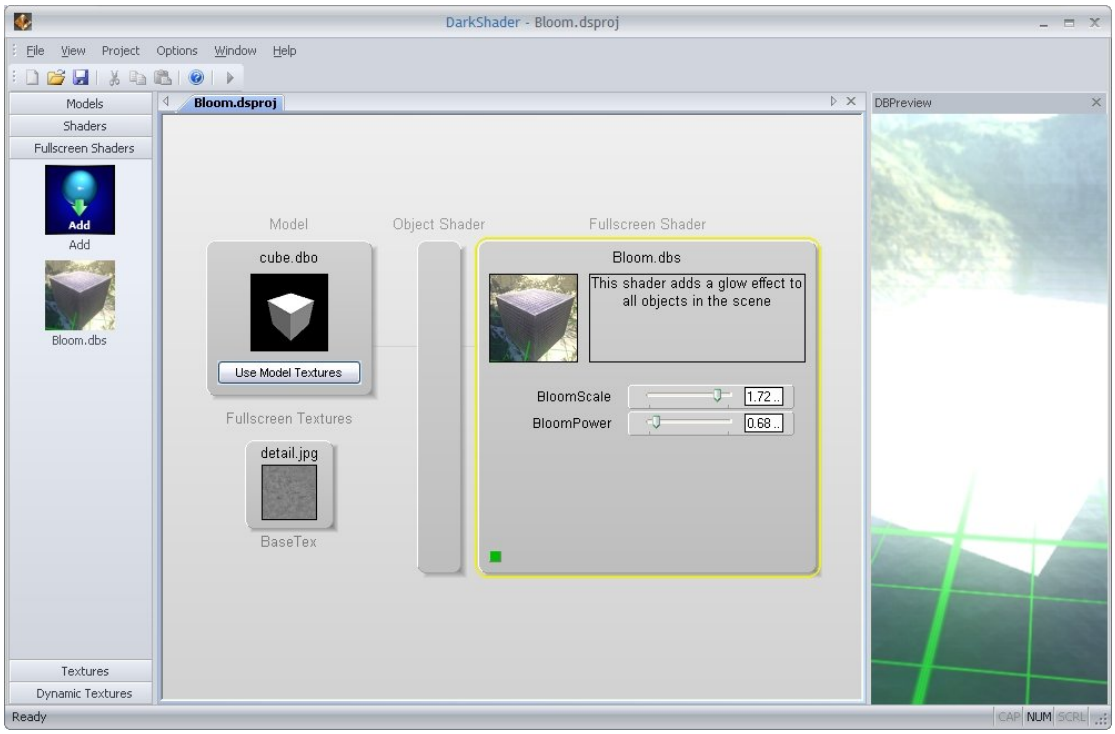
From this point onwards you can build shaders and experiment with all kinds of settings. However, for our purposes we're going to use Dark Shader in a very simple way. We will load one of the existing projects and select the Bloom shader:



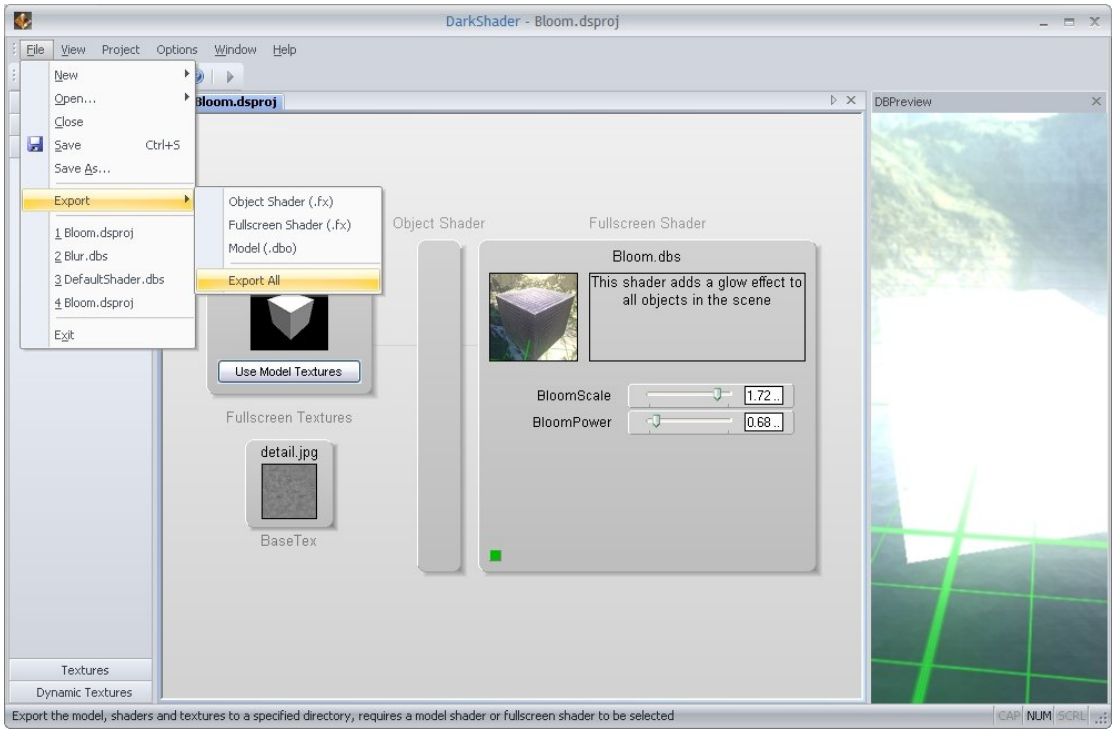
This shader will add a glow effect to objects in the scene. When the shader has been loaded we can start modifying its properties. For this shader the two main properties are the bloom scale and bloom power. The following image shows the default settings:



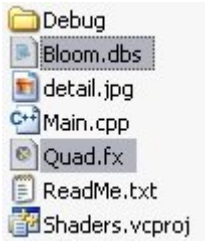
By moving the sliders to control the properties we can increase the bloom effect:



This will be good enough for our example. To export it we simply go to the File menu and select the Export option and then click on Export All.

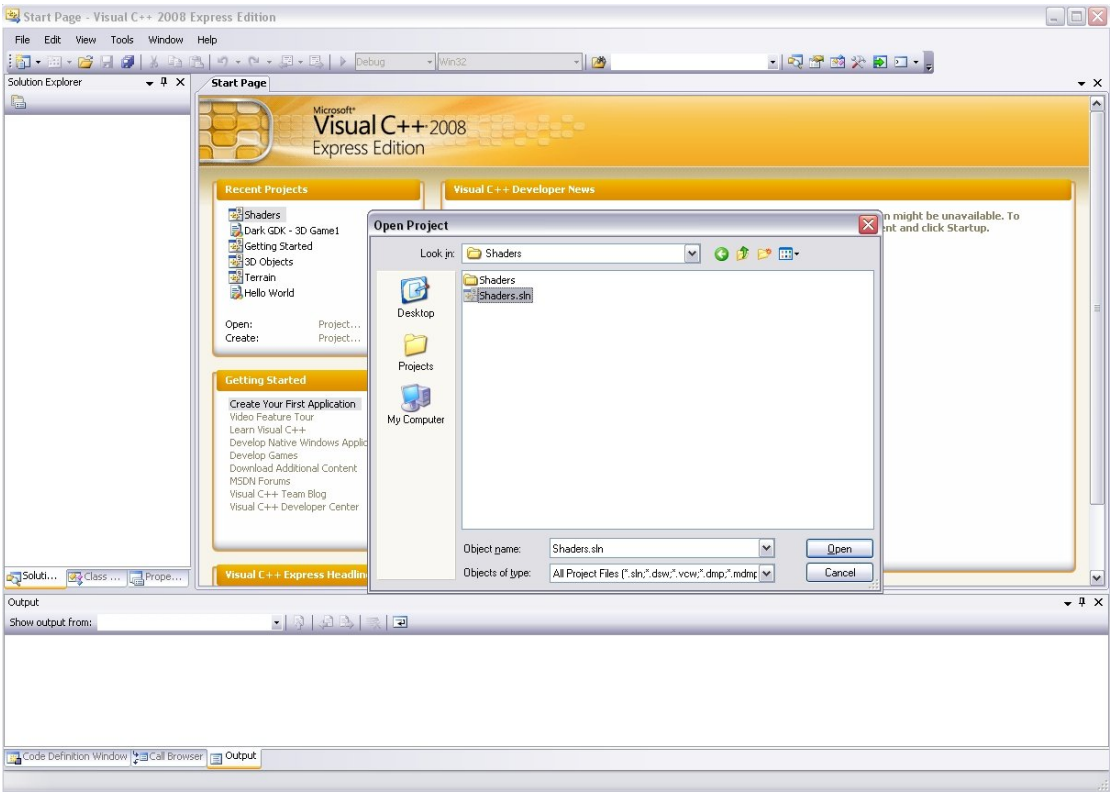


The export option creates a collection of files that will be saved out to the folder you specify. The main files we are interested in are the DBS file that contains the shader instructions alongside the FX file that also has shader instructions. These two files have been placed alongside our existing project files:

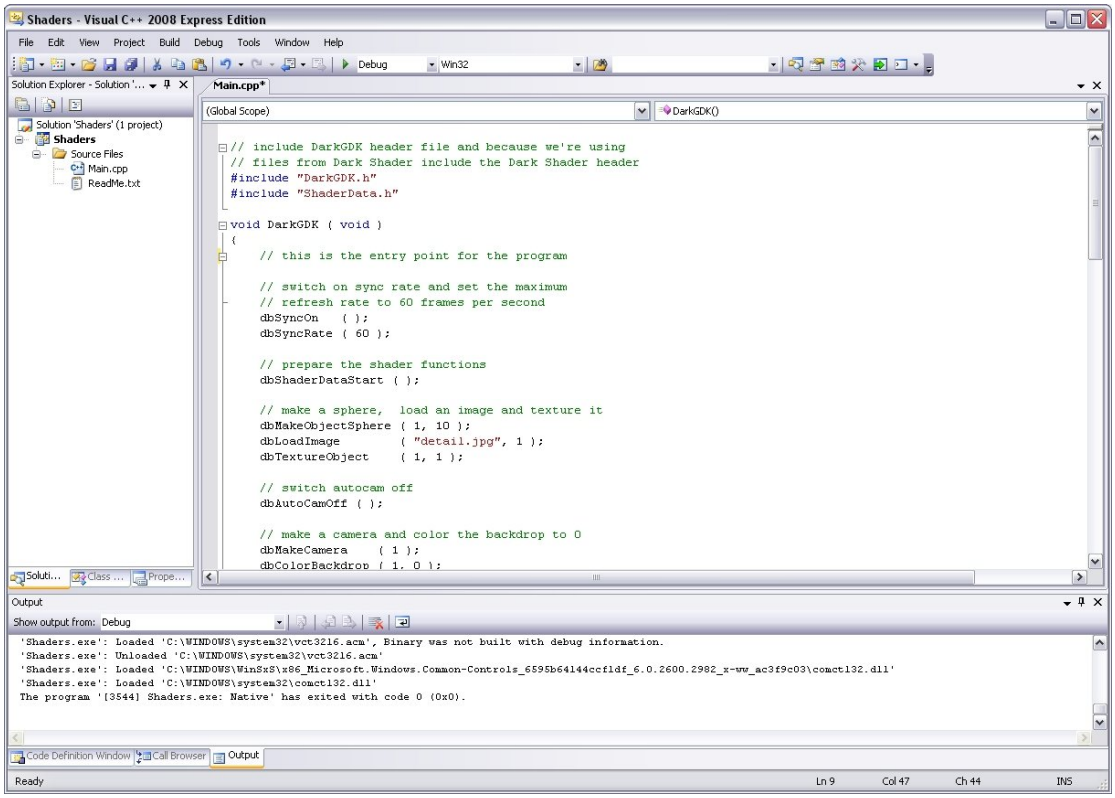


Opening the project

Launch Microsoft Visual C++ 2008 and 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 folder. Finally open the solution file for Shaders



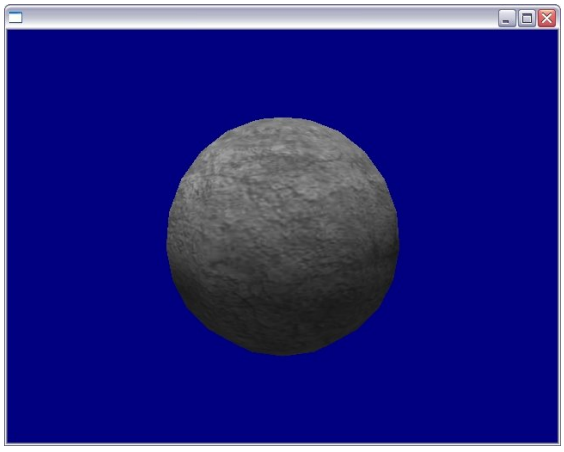
When the project has been opened you can see it has one source file – Main.cpp:



Techniques used in this program
Typically a program will draw the contents of our scene directly to the screen. This works perfectly well in most cases however, for our purposes it means we cannot utilise the bloom shader. This is because the bloom shader we have created works by taking an image and modifying its pixels to create a glow effect. This means we'll have to change the way our program works as we want this shader to be applied to all objects in our scene. This process may take some getting used to but once up and running it works really well and can be used for many different effects. Basically what we need to do is draw our scene but not directly to the screen. One method of doing this is to create a new camera and make this camera output to an image. Then later on this image can be manipulated by the shader and we can then display this image on screen.

To help in understanding this process we can examine a few basic examples. This first example is typical of many GDK programs – general properties are set, objects made and a main loop with a call to update the screen:

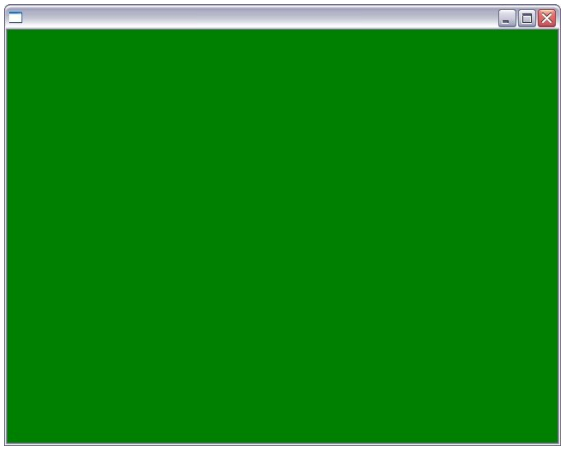
```
void DarkGDK ( void )
{
	dbSyncOn   ( );
	dbSyncRate ( 60 );

	dbMakeObjectSphere ( 1, 10 );
	dbLoadImage        ( "detail.jpg", 1 );
	dbTextureObject    ( 1, 1 );

	while ( LoopGDK ( ) )
	{
		dbSync ( );
	}
}
```



If you have followed the previous tutorials this code should be very familiar to you. Now what would happen if we introduced a new camera into the scene by calling the function dbMakeCamera? This particular function takes one parameter and that is an ID number for the camera. When your program starts a camera is automatically made and will be given an ID number of 0.

This default camera will be used all the time unless you make a new camera. In the following code a new camera is made:

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

	dbMakeObjectSphere ( 1, 10 );
	dbLoadImage        ( "detail.jpg", 1 );
	dbTextureObject    ( 1, 1 );

	dbMakeCamera ( 1 );

	while ( LoopGDK ( ) )
	{
		dbSync ( );
	}
}
```

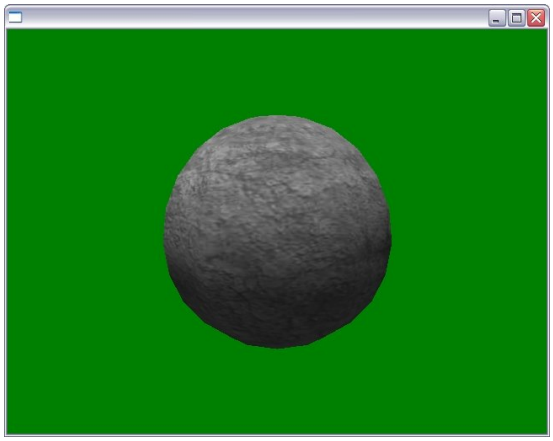


The results of this program may be completely different to what you expected. Instead of seeing the sphere on a blue background we see a completely green screen. This happens because after making our camera it becomes the main camera in use and by default our new camera will be positioned at 0, 0, 0 and rotated at 0, 0, 0. Going back to our first program you will notice the camera is in front of the sphere – this happens because by default the initial camera will be automatically moved in front of any new objects that get created. This process will not happen with newly created cameras. Another change is that new cameras will be given a green background as we can see in the screenshot.

If we positioned and rotated our new camera to match the original camera then we would get to see the sphere:

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

	dbMakeObjectSphere ( 1, 10 );
	dbLoadImage ( "detail.jpg", 1 );
	dbTextureObject ( 1, 1 );
```

```
	dbMakeCamera ( 1 );

	dbPositionCamera ( 1, dbCameraPositionX ( 0 ),
dbCameraPositionY ( 0 ), dbCameraPositionZ ( 0 ) );
	dbRotateCamera ( 1, dbCameraAngleX ( 0 ), dbCameraAngleY ( 0 ),
dbCameraAngleZ ( 0 ) );

	while ( LoopGDK ( ) )
	{
		dbSync ( );
	}
}
```

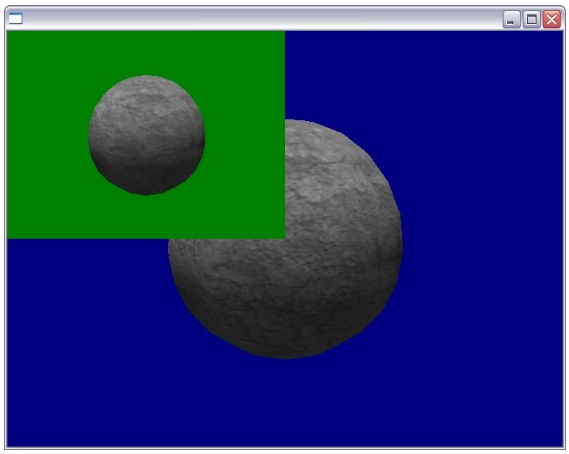


Now both cameras are matched in position and rotation so when our new camera is active we get to see the same results as with our original camera. Now you may be wondering what has happened to our first camera – well it is all still there but the new camera is obscuring it. This happens because whenever a new camera is created it will fit the whole screen. This can be altered with a call to dbSetCameraView. This function allows us to resize the camera and make it use only a portion of the screen. It takes four parameters defining left, top, right and bottom coordinates of a rectangle. Lets try using this to make our new camera fit into the top left of the screen:

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

	dbMakeObjectSphere ( 1, 10 );
	dbLoadImage ( "detail.jpg", 1 );
	dbTextureObject ( 1, 1 );

	dbMakeCamera ( 1 );
	dbSetCameraView ( 1, 0, 0, 320, 240 );

	dbPositionCamera ( 1, dbCameraPositionX ( 0 ),
dbCameraPositionY ( 0 ), dbCameraPositionZ ( 0 ) );
	dbRotateCamera ( 1, dbCameraAngleX ( 0 ), dbCameraAngleY ( 0 ),
dbCameraAngleZ ( 0 ) );

	while ( LoopGDK ( ) )
	{
```

```
            dbSync ( );
      }
}
```

With the new code after the camera is made the function dbSetCameraView is called. The ID number of our new camera is passed in and then we set the top to be 0, the left to be 0, the bottom to 320 and the right to 240. This results in our new camera being shown in a small portion of the screen at the top left:



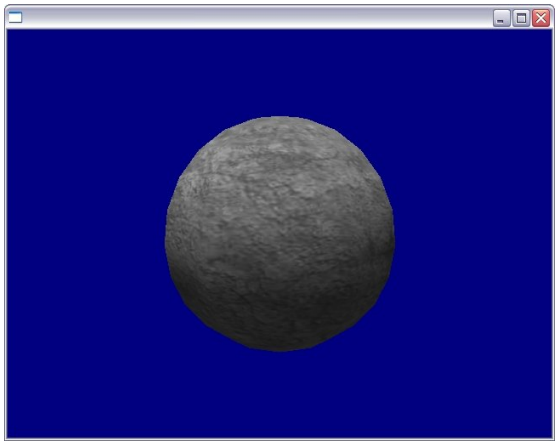
This change also allows us to see the original default camera.

In the next test we can output the second camera to an image. To achieve this we can call dbSetCameraToImage. This function takes several parameters. The first is an ID number for the camera, then an ID number for the image and finally the width and height of our image. In our code we can take out the call to dbSetCameraView as it is not necessary now are going to output the camera to an image. This line will be replaced with a call to dbSetCameraToImage. The first parameter is the ID for our camera, then we can use a value of 2 for our image ID as this is unused, then we'll pass in 256, 256 to create a 256 x 256 texture:

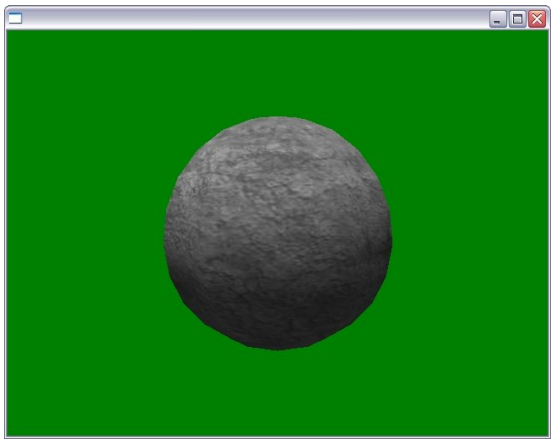
```
void DarkGDK ( void )
{
      dbSyncOn   ( );
      dbSyncRate ( 60 );

      dbMakeObjectSphere ( 1, 10 );
      dbLoadImage ( "detail.jpg", 1 );
      dbTextureObject ( 1, 1 );

      dbMakeCamera ( 1 );
      dbSetCameraToImage ( 1, 2, 256, 256 );

      dbPositionCamera ( 1, dbCameraPositionX ( 0 ),
dbCameraPositionY ( 0 ), dbCameraPositionZ ( 0 ) );
      dbRotateCamera ( 1, dbCameraAngleX ( 0 ), dbCameraAngleY ( 0 ),
dbCameraAngleZ ( 0 ) );

      while ( LoopGDK ( ) )
      {
            dbSync ( );
```

```
        }
}
```

Now lets see the results of our change:



The changes result in us seeing the default camera which shows the sphere. Our new camera is no longer displayed on screen and the reason for this is that it is being output to a texture. Our texture is in memory and ready to be used but as of yet we haven't done anything with it. Our camera will still contain the same contents but it simply in memory and no longer on screen.



For our next test we're going to create a new object and texture it with the image that has been created for our camera. To do this we'll create a cube, position it so it's out of the view of our new camera, position the default camera in front of this cube and then texture our cube with the image from the camera as follows:

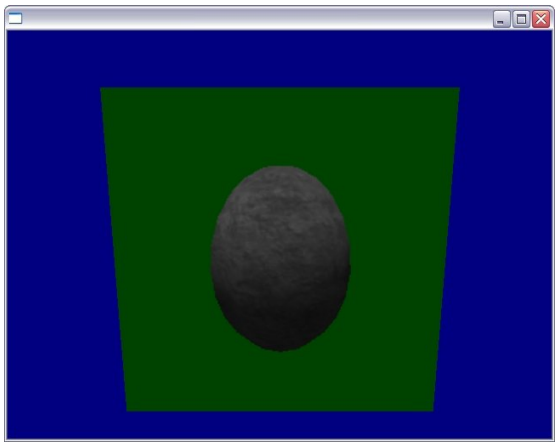
```
        dbMakeObjectCube ( 3, 10 );
        dbPositionObject ( 3, 100, 0, 0 );
        dbPositionCamera ( 100, 2.5, -15 );
        dbTextureObject  ( 3, 2 );
```

This new block of code is placed into the program just before the main loop:

```
void DarkGDK ( void )
{
```

```
	dbSyncOn   ( );
	dbSyncRate ( 60 );

	dbMakeObjectSphere ( 1, 10 );
	dbLoadImage ( "detail.jpg", 1 );
	dbTextureObject ( 1, 1 );

	dbMakeCamera ( 1 );
	dbSetCameraToImage ( 1, 2, 256, 256 );

	dbPositionCamera ( 1, dbCameraPositionX ( 0 ),
dbCameraPositionY ( 0 ), dbCameraPositionZ ( 0 ) );
	dbRotateCamera ( 1, dbCameraAngleX ( 0 ), dbCameraAngleY ( 0 ),
dbCameraAngleZ ( 0 ) );

	dbMakeObjectCube ( 3, 10 );
	dbPositionObject ( 3, 100, 0, 0 );
	dbPositionCamera ( 100, 2.5, -15 );
	dbTextureObject  ( 3, 2 );

	while ( LoopGDK ( ) )
	{
		dbSync ( );
	}
}
```

The result is as follows:



We have made a cube, repositioned the default camera and textured this cube with the image coming from camera 1. Note that camera 1 is still in front of the sphere – we haven't altered its position so as is to be expected the image on this cube is of camera 1 facing the sphere. This is the key to what we're going to do in our shader program. We will be doing a very similar process – outputting our scene to an image, this image will get modified by the shader and finally it will be drawn on screen.

Overview of code

The code for our shader program is going to use some of the camera techniques we have just seen. Before we delve into all of the code it's helpful to get a brief overview of what is taking place.

To begin with, general properties of the program are set such as sync rate. We then move onto setting up the shader library, creating a sphere, and

texturing it for our scene. Once we get past this point we start entering into the code that deals with shaders.

Due to the way our bloom shader works it is necessary to take the contents of our scene and output it to a texture. Our shader will then modify this texture to apply the bloom effect. The final result is an image that can be displayed on screen. To implement this process is not particularly complex. It is just a matter of dealing with multiple cameras and understanding where objects are being drawn to.

We need some method of taking this output image and displaying it on screen. To achieve this we can make a simple plane (quad) object, texture it with the image and then position it in front of the camera so that only the plane is drawn to the screen.

This whole concept takes some getting used to but once you've seen it in action it should start to become much clearer.

Our code will perform the following actions:

- create a sphere and texture it

- make a new camera
- load our bloom effect and apply it to our newly created camera, also define an ID so it can output to an image

- make a plane, a plane (often called a quad) is a very simple object, imagine a cube, it has 6 sides, now take just one side and that is the shape of a plane
- apply a special shader effect to it
- texture the quad with the image created for the bloom effect

- in our main loop, first stage is to hide the quad and show the sphere and output this to camera 1
- second stage is to hide the sphere and show the plane
- finally render the whole scene which at this point will simply be the plane

Now we have an idea of what is going on we can examine each part of the code in more detail.

Stage 1 – Initial Code
The first line of code in our program will include the Dark GDK header file. Directly below this we have another include this time for "ShaderData.h". This is a header file that comes with Dark GDK and it is necessary to include this file when using shaders exported from Dark Shader.

```
dbSyncOn ( );
dbSyncRate ( 60 );
dbShaderDataStart ( );
```

The first few lines inside the DarkGDK function are used to set the sync rate on and set the maximum update to 60 frames per second. Next we have a call to dbShaderDataStart. This is a function that needs to be used in conjunction with shaders from Dark Shader. It is essential that this function is called before dealing with any shaders as later on we're going to use some specific functions that rely on this initial call.

Stage 2 – Making our scene
Over the next few lines we create our scene – all we include is a simple sphere. A sphere is created and then an image loaded and applied to it.

```
dbMakeObjectSphere ( 1, 10 );
dbLoadImage ( "detail.jpg", 1 );
dbTextureObject ( 1, 1 );
```

Stage 3 – Cameras
In this part of the code there are two sections. The first is a call to dbAutoCamOff. This is used because by default Dark GDK will reposition the camera whenever an object is loaded. For example, if you make a sphere the camera will get placed directly in front of it. Later on say you make a box the camera will get moved in front of the box. This is useful in some circumstances but for what we are doing it is not necessary so we simply switch it off by making a call to dbAutoCamOff. The end result is that we are in complete control of the camera.

```
dbMakeCamera    ( 1 );
dbColorBackdrop ( 1, 0 );
```

The next two lines include a call to dbMakeCamera and dbColorBackdrop. The dbMakeCamera function takes one parameter and that is an ID number. When your program starts camera 0 will get created automatically and will be the default camera. If you want to create a new camera you must start from an ID of 1 or above. The call to dbColorBackdrop allows us to change the backdrop of the camera. Whenever a camera is created the initial backdrop colour will be green but for our purposes we change it to black by passing in the ID number of the camera and then a colour value of 0. Say we wanted to change our backdrop to red we could pass in dbRgb (255, 0, 0).

Stage 4 – Bloom Effect
This block of code has two function calls. The first is a call to dbLoadCameraEffect. This function will take an effect file and apply it so it will be used on a specific camera. It takes three parameters. The first parameter is a filename for the shader. When we exported our shader "bloom.dbs" was created so this is passed in. The second parameter is an effect ID number. These numbers are completely separate from other ID numbers therefore you can have an object loaded into ID 1 and an effect loaded into ID 1. In our call we use an ID of 1 for the effect. The final parameter controls whether textures are loaded for the shader. This is useful in instances where a shader relies on a certain texture that gets exported. It is not relevant for our demo so we simply pass in a value of 0 and ignore it.

```
dbLoadCameraEffect ( "Bloom.dbs", 1, 0 );
dbSetCameraEffect  ( 1, 1, 1 );
```

The second call in this block of code is to dbSetCameraEffect. This function applies an effect to a specific camera and allows you to specify an image to output it to. It takes three parameters. The first is an ID number for the camera. We earlier created a camera with an ID number of 1 and we want to apply our effect to this camera. Therefore we pass in a value of 1. The second parameter refers to the effect ID. Previously we call dbLoadCameraEffect and loaded our bloom shader into an effect ID of 1 therefore we pass in a value of 1 for this parameter. The final parameter is used to define an image ID – the effect will output to this image. You can use any ID number that is not already in use for an image. We currently don't have any image loaded into ID 1 so we use this ID.

Stage 5 – Plane object

In this stage we create a plane object. This object will be used to display our final scene output. The first step is a call to dbMakeObjectPlane. This makes a simple plane (quad) object and gives it an ID number of 2 and a size of 2 on the X and Z axis. This size is important because the quad.fx shader will use it to stretch the plane across the screen when rendering.

```
dbMakeObjectPlane ( 2, 2, 2 );
dbLoadEffect      ( "quad.fx", 2, 0 );
dbSetObjectEffect ( 2, 2 );
```

The second line is a call to dbLoadEffect. This function will take our shader named "quad.fx" and apply it to our object. The function takes three parameters. The first is a filename for the shader, the second is an effect ID and the final parameter defines whether any default textures are loaded. We pass in "quad.fx", use an effect ID of 2 and ignore default textures.

Stage 6 – Setting properties for our plane

This block of code will deal with passing data into our "quad.fx" shader. The shader relies on this data to operate correctly. When dealing with shaders it is very likely that you will need to pass in data to them.

The very first line calls the function dbMakeVector4. This is one of the many functions contained within the 3D mathematics function set. It is a simple method of exposing vectors to the end user. Again just like with many other GDK functions it relies on an ID number. The ID numbers for this function set are separate from others e.g. Basic 3D. A vector4 object is a simple way of storing 4 values. This initial call will make a vector4 object and give it an ID number of 1.

In the second line the vector4 object has its data set. Several values are passed into this function. The first is the ID number of the vector. The second is the width of the screen, the third is the height of the screen and the two remaining parameters are not required so values of 0 are passed.

In the third line a call is made to dbSetEffectConstantVector4. This function will effectively take the data we made in the previous call and pass it into a property of the shader named ViewSize. The first parameter is an ID number of the effect, the second is the property name and the third is the ID number of the vector. The call following on from this will delete our vector4 object as it is no longer required.

```
dbMakeVector4 ( 1 );
dbSetVector4 ( 1, dbScreenWidth ( ), dbScreenHeight ( ), 0, 0 );
dbSetEffectConstantVector ( 2, "ViewSize", 1 );
dbDeleteVector4 ( 1 );
dbTextureObject ( 2, 0, 1 );
```

One remaining call in this block of code is to dbTextureObject. What we are doing in this line is saying we want to texture our plane object with the texture that is being output from our bloom shader effect. The function takes three parameters. The first is an ID number of our object, the second is a stage number, we use a value of 0 to manage the default stage, finally an ID number is used to define the image. Our image was created with an ID number of 1 so we pass this value into this parameter.

Stage 7 – Camera properties
The final step before the main loop is to ensure both cameras are positioned and rotated in the same way.

```
dbPositionCamera ( 1, dbCameraPositionX ( 0 ), dbCameraPositionY ( 0
), dbCameraPositionZ ( 0 ) );
dbRotateCamera ( 1, dbCameraAngleX ( 0 ), dbCameraAngleY ( 0 ),
dbCameraAngleZ  ( 0 ) );
```

Stage 8 – The main loop
There are a few steps that need to take place inside the main loop. To create some movement the sphere is rotated on the Y axis by calling dbTurnObjectLeft. We could have also called dbRotateObject or dbYRotateObject to achieve the same effect.

The next block of code hides object 2 which is our plane, it then shows object 1 which is our sphere and last of all calls dbSyncCamera. This function will render our camera that contains the effect and in doing so prepare image 1 to be displayed on screen.

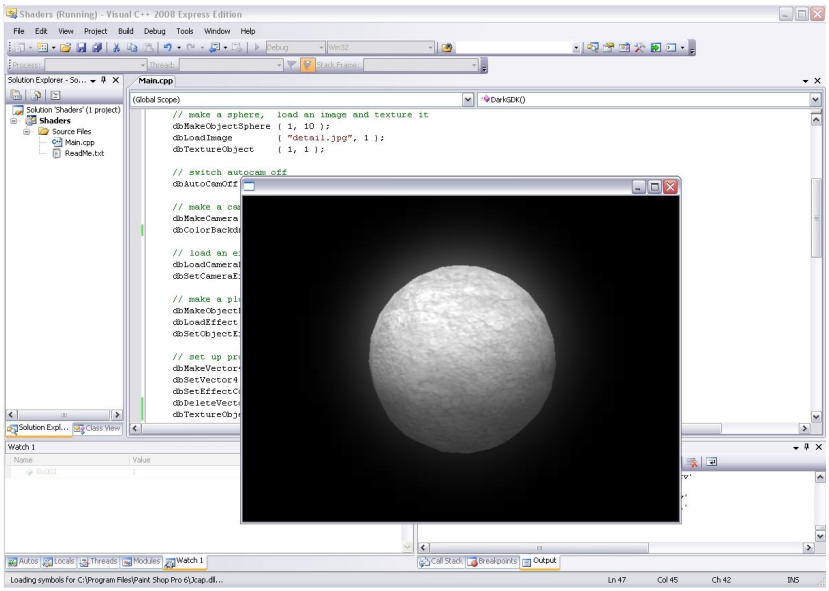
The next few lines of code hide object 1 and show object 2. This is done as we're now preparing to show our plane on screen so it isn't necessary to draw our sphere again.

The final call in the main loop is to dbSync. This will refresh the screen and update its contents.

```
while ( LoopGDK ( ) )
{
      dbTurnObjectLeft ( 1, 0.1 );

      dbHideObject ( 2 );
```

```
        dbShowObject ( 1 );
        dbSyncCamera ( 1 );

        dbHideObject ( 1 );
        dbShowObject ( 2 );

        dbSync ( );
}
```

Finally we're at the point where you can compile the program and see the results:



Conclusion
Using Dark Shader and a few lines of code we've been able to implement a bloom shader effect.

Before leaving this tutorial you may find it beneficial to experiment with the code to try and gain a full understanding of what is going on.

Try setting the wireframe state of the plane to 1. This will help you to understand how the final scene is being rendered to this plane which is being stretched over all the screen.

Remember back when we created the shader in Dark Shader. It had two properties that we could alter – BloomScale and BloomPower. We can alter these properties at runtime by using the function dbSetCameraEffectConstantFloat e.g.

```
dbSetCameraEffectConstantFloat ( 1, "BloomScale", 2.0f );
dbSetCameraEffectConstantFloat ( 1, "BloomPower", 0.60f );
```

Try adding these two lines directly before or inside the main loop and see the difference it makes to the shader.