Tutorial

Tutorial 1: The Basics!

Revision 1.0

mCODE AS

Introduction

This tutorial will teach you the very basic of mBLAZE3D to help you get started using this programming tool. We will create a very simple application with a scene consisting of 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 mBLAZE3D works!

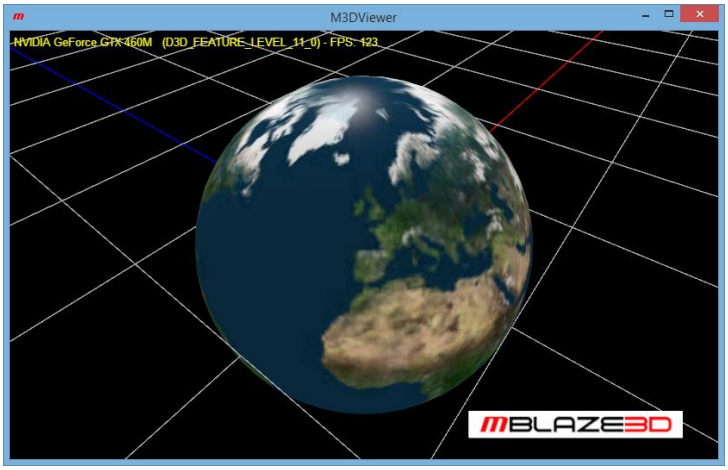

Figure 1 The resulting application.

Tutorial

Step 1

The first thing we have to do is to start up mBLAZE3D Studio. This should look something like Figure 2. The different elements of the Studio are explained in the user manual, so we should not talk too much about it here.

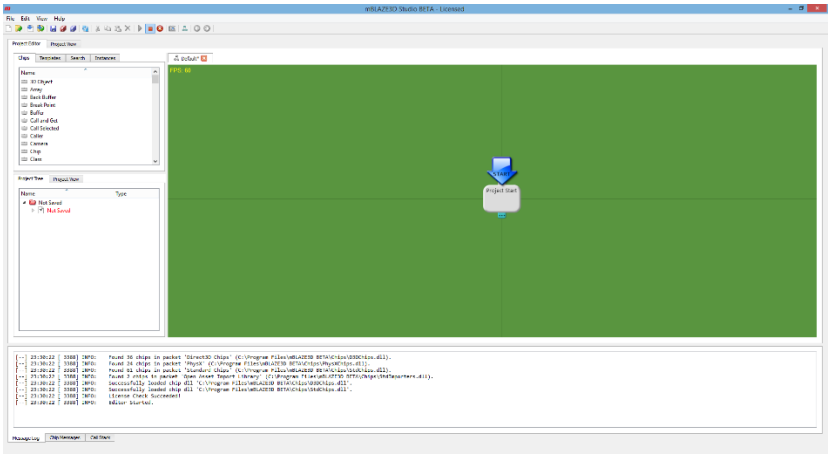

Figure 2 mBLAZE3D Studio.

Step 2

We are now ready to start programming our first application! mBLAZE3D is all about adding, configuring and connecting different type of *chips* in workspaces called *classes*. The green area, the *editor view*, in the Studio is at startup a default, empty class added for your convenience! It only contains one single

chip, set to be the *start chip* for the project. If you have a programming background, think of the start chip as the *main method* for your program!

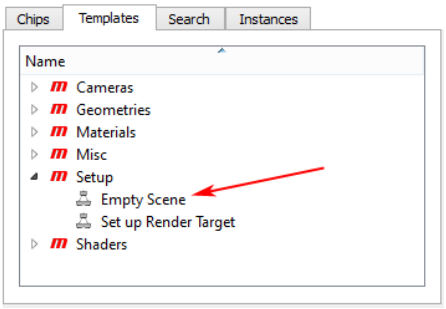

Figure 3 The template panel.

We now want to add more chips to our class to construct the program logic. We could do this by adding individual chips, or we could use *templates* for adding common functionality. A template is just a collection of chips and connections, defining functionality often used. They are very convenient because we don't have to construct the functionality our self every time we need it! The *templates panel* in the upper left corner of the Studio, as shown in Figure 3, contain all available templates. We want to create a simple scene, so the template for an empty scene is just what we want. Find the 'Empty Scene' template located in the 'Setup' document, press your left mouse button on the item, drag it into the editor view, and drop it at a suitable location below the start chip to get a situation looking something like Figure 4. You should now connect the upper `Caller` of the inserted template to the start chip, by dragging a connection from the `Caller`'s top connection to the child connector of the start chip.

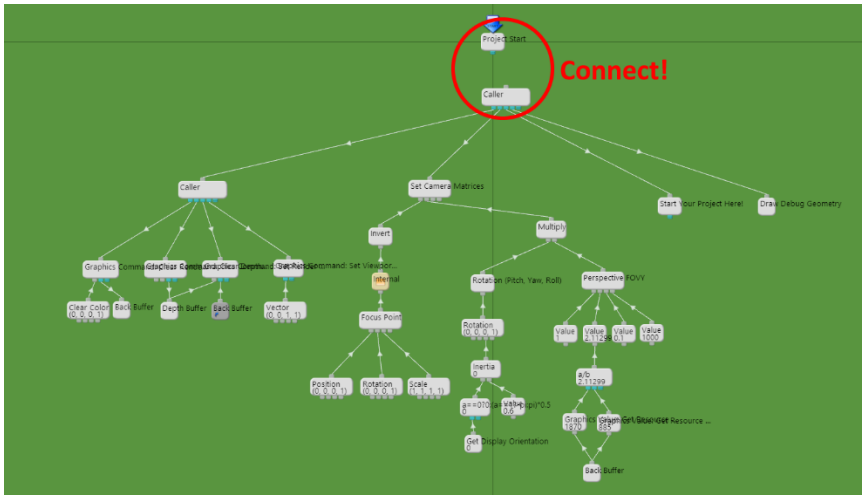

Figure 4 Adding the 'Empty Scene' template.

Step 3

The next step is to try running your application in one of the two project views. You can do this by selecting the *Project View* tab in the upper left corner of the Studio to get the large view, or in the lower left panel to get a smaller view. The benefit of the smaller view is that you can keep working in the editor view at the same time as your project is 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.

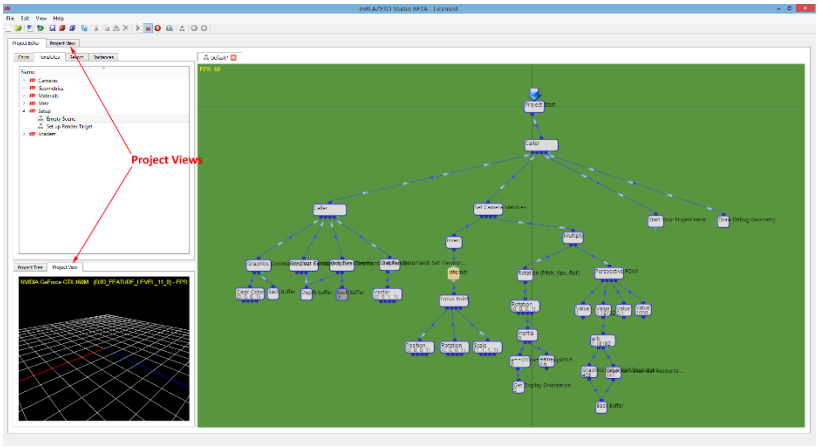

Figure 5 The Project Views.

What is now drawn to the view as shown in Figure 5 is the result of the logic in the template we added to the project. Active parts of the code is highlighted when a project view is open.

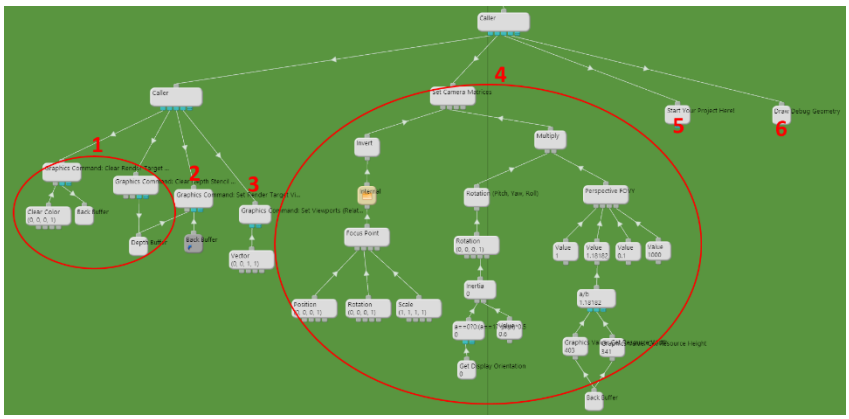

Figure 6 The different elements of the 'Simple Scene' template.

As indicated in Figure 6, the template added the following functionality:

1. Clears the *back buffer* (The window we render to) and the *depth stencil view texture*.
2. Sets the back buffer and the depth stencil view as the current *render target*.
3. Defines a *viewport* to cover the entire size of the render window.
4. 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 by the user by mouse or touch screen. The logic for this hidden in the 'Internal' folder. Try have a look inside and see if you can understand what is going on!
5. Defines a starting point for you to add additional logic.
6. Render debug geometry like the world grid.

Step 4

So far, we only have an empty scene without any objects in it! That's exactly what we are going to add now, and we use a template for that as well, namely the 'Sphere' template, located inside the 'Geometries' document. Try adding one and connect it to the 'Start your project here' chip from the previous template. You should now see a white sphere in the center of your scene, right?

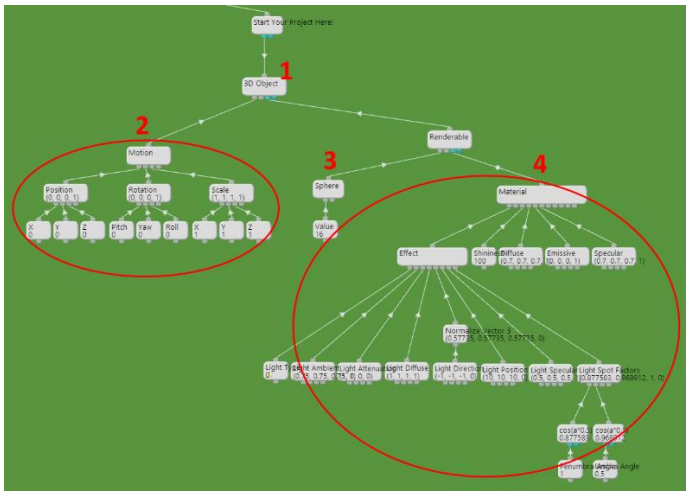

Figure 7 The 'Sphere' template.

The template you just added defined a 3D-object. It may look a bit complex, but let's have a look at the different elements as indicated in Figure 7:

1. This is a `3D object`, requiring a `Matrix` for position and orientation, and some `Renderable`s linking `Drawable`s (3D-geometry) to materials.
2. The matrix is in this case defined using a `Motion`, making it easy to set position, rotation and size for your object. Try changing the X, Y and Z coordinates for the position by double clicking the values to bring up the property dialogs, and see how you can move the object around.
3. The 3D-geometry is in this case defined by a `Primitive`. This chip contains a set of simple shapes you can choose from in the property dialog, among them the sphere.
4. The material system of mBLAZE3D is very generic! No things are predefined, so you will have to set up everything yourself, including material attributes and properties, lighting and shading etc. This reflects the concept of Direct3D 10/11 where there is no fixed pipeline with predefined material and lighting properties as is was in the 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, but again templates are here to help you out! The 'sphere' template you added included a standard material and effect system. The `Effect Material` defines a set of properties that is 'unique' to the geometry it is linked to. The `Effect` contains shader code that actually determines how the material and its properties are to be rendered. It is common to share an `Effect` between many `Effect Material`s. You could for example have an `Effect Material` defining a red surface, and another one defining a green surface, both rendered using the same `Effect`. In our case, the material defines a set of common material properties, and the effect implements a standard way of rendering them using one single light source. It's very basic, and something you probably want to extend in a real application!

Step 5

The next step is to add a *texture* to the sphere! A texture is as most other things in mBLAZE3D implemented as a chip. Go to the *list of chips* (next to the templates panel), find the `Texture` chip, and drag in into the editor view somewhere below the material chip. Connect it to the material's 'DiffuseTexture' child connector. By default, this material is configured not use this texture, so we have to open its property dialog as shown in Figure 8, and change the value of 'DiffuseTexEnable' from 0 to 1 for the effect to actually include the texture in the shader code. Don't worry, it's not as complicated as it may look like!

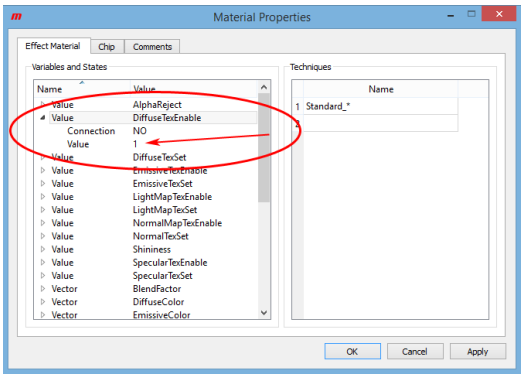

Figure 8 The material property dialog.

Now we have to find an image to use as a texture. Open the Texture's property dialog, and click the 'Load Image From File' button. Select the 'earth.jpg' file located in the Tutorial folder of the Studio's install directory. By default, mBLAZE3D uses a *sRGB* (google is your friend!) back buffer. 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 compression as shown in Figure 9! These cryptic terms are used in the Direct3D API and should therefore be familiar to people with a background from Direct3D programming. It has been the goal to make things recognizable to experienced programmers, and to expose as much functionality from Direct3D as possible, without leaving out stuff that advanced users could find useful, just for the sake of simplifying things. Hopefully, it will still be manageable to beginners as well. After all, this is a tool packed with powerful features, supposed to be convenient to non-programmers as well as hardcore software developers!

Okay, looking at the project view, the sphere should now have a nice earth texture applied to it!

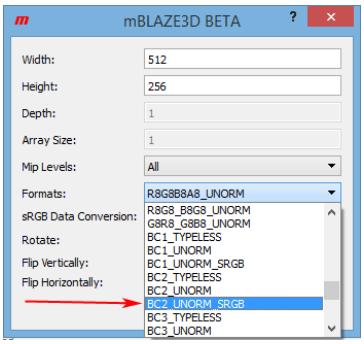

Figure 9 Texture import dialog.

Step 6

Now we should try making the world go around! If you guessed we should now look at the 'Rotation' component of the Motion chip, you were right! The rotation is given as *euler angles*, and we should make the sphere spin around the Y-axis (up), that is the *Yaw* component. An `Expression Value` is perfect for this! Find one in the list of chips, add, and connect it below the Yaw value. An `Expression Value` is used for writing simple mathematical expressions. Open the property dialog and enter 'old+dt' in the expression box, as shown in Figure 10. The 'old' component is the value the chip currently holds. 'dt' is the time the last frame took to render. 'old+dt' is therefore a clock counting seconds. In our case, the value is interpreted as an angle in radians, so the rate of change is *radians pr second*. If you want the sphere to spin at another rate, you can multiply the 'dt' component by some factor, e.g. 'old+dt*2'. If you want to go one step further, you can extract the factor as a value linked to the expression's child connector, then change the formula to 'old+dt*a' where 'a' is the first `Value` connected. ('b' is the second and so on). Now, look at the project view. Pretty cool, right?

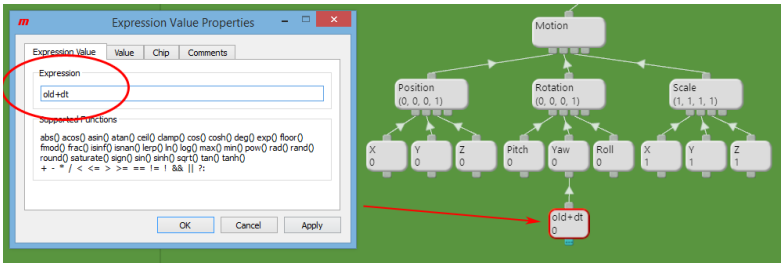

Figure 10 The Expression Value.

Step 7

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

NOTE: If you are using a trial license, you will not be able to continue from here! You can register your copy of mBLAZE3D using the license wizard inside the Studio to receive a free license key unlocking the save functionality, making it possible to publish your own projects!

The first thing we have to do is to save the document containing our class to some file. Select 'Save' from the toolbar or the file-menu to bring up a file dialog to 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 change anything manually!

To publish the project, we have to select 'Publish…' from the file-menu. This will bring up the *publish wizard* to configure the resulting application. First, we will focus on the target platform set in step 3 as shown in Figure 11. If you are running a 32-bit version of Windows, you should select 'Windows Desktop Application (x32)' from the list of options. Similar for the 64-bit version.

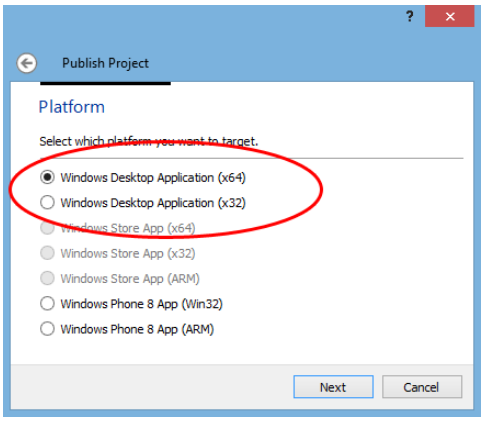

Figure 11 Selecting target platform.

Step 4 is about selecting the format of the published project. We will go with the default, which is a *self-extracting executable archive*. Verify the target file name and location as indicated in Figure 12 before proceeding. The next step is about selecting which project files to include with the publication. Our file should be selected by default, so you can just go 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 publication!

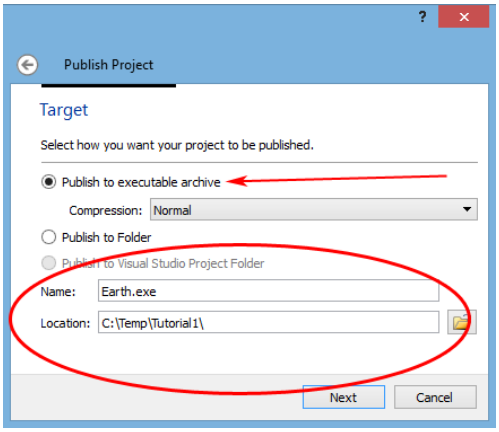

Figure 12 Selecting format and target location.

Now, use your file browser to find the new exe-file you just created. It's probably located in the folder where you saved your project file. Start it and be amazed! 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 to see what effect it has. That's the best way to learn how to program in mBLAZE3D!

Enjoy!