Exporting models from Maya to Virtools, especially in the context of the chess scene is simple, but requires a little bit of subtlety. There are two ways to do it, and you should choose between them depending on the type of geometry you are working with. The first method is what you use when you have a single mesh, that is a polygon object for which all of the geometry is in one single polygon object. The second technique you can use if you have an object which has multiple things grouped together.

In some situations, you can easily take all of the polygon geometry for separate objects, combine them, and then export the combined model into Virtools. Other times, especially when you have animated objects, the process is not so simple and the second method is necessary.

Exporting a single mesh

When you have a single mesh, the export process is very straightforward. Here I have modelled a funny single mesh based off of a cone.

On its own, the geometry is fine, but in order to make sure that the mesh does not sink into the board in the chess scene, you must move its pivot point to the base of the object. The pivot point is the focal point of the object, and in the chess scene, when objects move, they move so that their pivot point sits on the top of the chess boards. So if an object has its pivot point the center, it will sink so that it's halfway through the board when it moves.

To move the pivot point, select the translation tool, then tap the Insert key on a PC, or the Home key on a Mac. Now you can move the pivot point around. When you are done, tap the Home/Insert key again.
It is okay if the object is not sitting on the origin, but it is important that the pivot be moved.

Replacing meshes in Virtools

Export the mesh normally, and now we'll bring it into the chess scene.

I've done this here in the chess scene. Behind the scenes, several things happened when the .nmo was imported. Virtools separates scene components into 3d objects, 3d frames, materials, meshes, and textures. When I imported the file, Virtools created three objects: a material "lambert1" (because I didn't put a new material on the object), a mesh called "pConeShape1", and a 3d object called
"pCone1". These objects are separate, but dependent on each other.

A mesh is the raw polygon geometry that defines a shape. In Virtools we are fortunate enough to be able to separate the mesh from the object itself. Thus, an object has a reference to a mesh, and depends on it, but the mesh can exist separate from the object. So, here the first thing that you should do when importing a mesh object into a scene is to delete the object, but not the mesh itself. When I delete the object Virtools will ask me whether I want to delete the dependencies. For this we will use "No Dependencies", since that way it won't delete the mesh, just the object.

The object disappears, but the mesh is still in the scene. Now we can select the object whose mesh we want to replace, and work with that. Right click on the object in question, and then select "3D Object Setup"
Make sure you are on the Parameter tab of the object setup panel. In the window in the lower left of the screen that says "Object Meshes", you can right click in the panel and select "Add Mesh", and then browse to the mesh that you just imported.

Then, right click on the previous mesh that is listed and select "Remove Mesh". Now the mesh is
replaced. The actual object for the pawn in this scene has remained essentially unchanged except for
the appearance of the mesh that is used.

AFTER you do this, you must click on the "Set IC" button to the left of the Object Meshes box. This
will make sure that when the scene rewinds to the beginning of the chess match, the new mesh stays
there, otherwise it will be reset back to the old version.

Because the mesh has been loaded, you don't need to import the nmo again, unless you want to update
the mesh. Thus, you can update all of the pieces that you want without ever needing to reimport the
same model.

Exporting several meshes grouped together

This second method requires a bit more involvement, but allows you to import much more complex
objects and do more interesting things with them. It requires a little bit of care in both the Maya and
Virtools sides of things.

I'm modelling something here that is a bit bigger and has multiple objects as a part of this piece. I'm
putting multiple materials and a texture on this so that it's a little bit more interesting to look at and you
can see some of the export challenges. It consists of 15 objects, which is a lot to handle when working
with all of them in Virtools.
Right now, my objects are all independent, and if I export this scene directly, it will include all of the objects separate in the scene, and it will be impossible to organize. So to improve that, we will put the whole thing into a group. First thing I am going to do is to move the objects up to so that they are situated on the origin, then select everything and press CTRL-g to group them together.

If you select objects directly in the scene, you won't be select the group itself, because it is invisible. So
in order to work with the group, you must use the hypergraph to explore the scene and work with it there. When you export, select the group node, and export that.

NOTE: There was a little popup when importing that which read "Duplicate Names Found" for "file1". This is a texture node, You should select "rename" on your first import so that there isn't a collision. More on this later...

Again many things happened behind the scenes on this import. Aside from the file name collision, there are now 15 new objects in the scene, for each of the geometry parts that I imported, as well as a strange 3D plus sign that is under my new geometry. This is a 3D frame, and it represents the group. If you select it and move it around, you will move around the entire structure, just as if you were moving the group in Maya.

In order to get the object to play nicely with the chess scene, we must take several steps to make sure it gets placed properly. I am going to replace "white_bishop_1" with the new structure, because it is one of the first major pieces to move. To do this, I am going to delete the object, but first I want to check its Attributes. Right click on the object to be replaced and select "3D Object Setup", then click on the Attribute button on the left side of the setup pane.
Specifically, we are looking for the "positionFrame" attribute, and its value, which in this case is "1f". Now, we can delete the cone that is there, and replace it with our imported object. Choose "No Dependencies" in the delete dialog. Be careful to move the new object via the 3d frame and not any of the sub objects. Use the spacebar to lock your selection if that will help.

Once the object is in place, select "Attribute" and click on the "Add Attribute" button at the bottom of the screen. Find positionFrame under the "No Category" item, and select "Add Selected", and then "Close". You've just added the position frame attribute, and now you can set its value to be the value for the piece we just replaced, "1f". Now the object knows what square it is on.
We're not quite done yet, there are a few more things that need to be done to make this thing ready. Navigate through the Level Manager, and find where the 3D Frame for your group is. The first is that you need to put the 3d frame into the group of "all figures". This group is the list of all of the chess pieces in the scene. This is what is necessary to make sure that the piece moves now. Next, you must set the initial conditions on the 3d frame, and also the sub objects.
Now, that one is pretty much finished. If things don't work properly, make sure that

1. The positionFrame is set on the 3D Frame
2. The 3D Frame is in the group of all_figures
3. The IC on the 3D Frame is set
4. The IC is set on the sub objects for the 3D frame

After all of that, it should be OK.
Importing Another One

Okay, suppose you have one imported, if you want to bring in another one, what do you do? The process is similar to the first, but there are some details that you should be aware of. In the following, I'll replace the second white bishop who moves again towards the end of the scene.

In order to do this, you can't do the same thing with the simple replacement that can be used with single mesh objects. Instead, you must re-import it so that Virtools will create the new scene objects properly. When you do this, Virtools will give you a variety of options. Generally it is best to "Use Current" for parts of the scene that already exist and you don't want to mess with. This is specifically true with textures and materials. For that you want to create new versions of, you must select "Rename".
Use **Current** for things that already exist in the scene

**Rename** objects that you want to copy
After that, you must repeat the process with removing the old object, setting positionFrame on the new 3D Frame, adding it to the group, and setting all of the initial conditions.

**Finally...**

Did you notice that the positionFrames for the bananas were reversed? The one that should be on 1b is in fact on 1g and vice versa. There are also a couple of bugs in the scene where pieces sometimes overlap. I am aware of these, but haven't worked out fixes yet.

---

**07: Camera Scripts in Virtools**

[Download the archive for this lab](#)

This lesson will cover how to create cameras and use camera scripts, specifically in the context of the Chess scene. This involves working with the Virtools building blocks a good deal, and hopefully it will
be a chance to help you get comfortable with them. The lab archive does not have any Maya files, but does have a version of the chess scene with an extra camera, and an importable version of the same camera.

It should be noted that these scripts do not need to be just for cameras, but may be used to trigger other sorts of events, such as sounds, particle systems, lighting effects, and many others.

Looking at existing camera scripts

In the chess scene, all of the actions and moves are defined in one array called "moves". If you find that array and go to the Array Setup, you can see a list of all of the moves in the chess game, and the cinematic scripts that are associated with them. Each row in the array contains a move, and an optional script that is executed when the move starts.

There are only 8 scripts in place now, and it's your job to finish the scene by adding additional scripts. Let us look at some of the first scripts and examine those in some detail. The scripts are, in order:

1. establishing shot1 Script
2. cam2 Script
3. cam2 FOV Script
4. white_back_cam Script
5. white_back_cam turn1
6. black_back_cam Script
7. topview Script
8. spline_camera Script

Here are the cameras and camera scripts that are in the scene. Each camera script is associated with a particular camera that exists in the world. They all have their initial conditions set so that when they are
moved over the course of the chess game, they will go back into place when the scene is reset. Note also that each of the camera scripts are not active at screen start. If they were, the camera scripts would play right away, but we want them to play when they are triggered through the main game loop.

establishing shot1 Script

The first script is "establishing shot1 Script". Let's look at this in the Schematic. The script should appear fairly simple. The first thing it does is to make the current camera (establishing shot1) the active camera. Then, it waits for some amount of time using the Timer building block (BB for short), and then it moves somewhere using the Move To BB. If you mouse over the input at the top it says "Duration (Time): 00m 01s 000ms", meaning that it will wait for one second before continuing. The MoveTo has parameters, but the most important ones are the Destination referential (which is where the camera will move to), and the Duration, which defines how long it will take.

Note that the Timer and Move To BBs have two pair of input and output connections. That is because these are loops. They get control from the top input, and will loop until their timer expires, when they will send control back to through the top output.
**cam2 Script**

The cam2 Script is also very simple. The script makes "black_back_cam" the active camera, then waits for a second, and then plays a sound. The sound that it is supposed to play was not actually exported properly with the scene, so there won't be a sound for this one.

**cam2 FOV Script**

Afterwards the cam2 FOV script is called, and gradually changes the field of view for the camera. This happens when the camera seems to pull back when the white bishop moves out. The camera is actually not moving, but its field of view is changing. Mousing over the inputs for the Bezier Progression BB, you will see that this loop is outputting an angle from 39.5° to 85° over 3 seconds. It is called a Bezier Progression because you can define a Bezier curve that changes the rate at which the values will change. The output leads from the bottom of the building loop block to the Set FOV BB. This actually sets the field of view on the camera.

You could think of this script as having the flow:
1) Go into Bezier Progression and set the timer to 0
2) Is my time up? If not, continue, otherwise break.
3) Get an angle in between 39.5° and 85° that is along the progression curve, based on my time.
4) Set the field of view to the number from step (3).
5) Goto (2).

**white_back_cam Script and white_back_cam turn1**

The first of these scripts isn't tremendously interesting, but the second one bears more promise. Both of these scripts are associated with the same camera, and notice that the second doesn't call "Set As Active Camera". It is not needed because the camera is already activated.

The second script uses the Look At BB, and uses that to loop back in on itself. This BB causes the camera to point at a specific target, in this case, white_bishop_1. The Following Speed parameter affects how fast it moves when it is set to look at the target. Its current value, 10% is enough to make a swift but smooth transition.
black_back_cam Script
The scripts get a little sneaky here. Note that black_back_cam is the same camera that was used for "cam2 Script" and "cam2 FOV Script". It's being used again here, and is being reactivated. Here we have a Curve Follow BB, but this particular building block is not being applied to the current camera, no if you mouse over the little square input on the left side of the BB, you'll see that it's applying the Curve Follow to "New Light.0001". The input with the square is called a Target Parameter, and defines what the building block is being applied to. The curve it is following is "New Curve", which is the curve that surrounds the chessboard.

topview Script
This is the camera that is on the top of the chessboard. This particular script does two separate things at once. The first is a Move To, which pulls the camera closer into the scene, and the second is a Bezier progression that widens the camera lens. The effect is a little wobbly, but illustrates how you can have multiple things happening simultaneously that are triggered by a script.
**spline_camera Script**

The final camera script is the one that breaks the scene. It may appear a little complicated, but it is not more so than the other scripts.

The first thing that happens is that the camera position is set to the first point on the curve. Then the camera is made active. This is useful so that the transfer doesn't cause a flicker when the scene runs. In this there is a Bezier progression which loops through a Position On Curve BB. The Curve Follow that is below the others doesn't do anything, as it is not connected to anything. The Position On Curve BB takes a percentage and sets the position for the object to be that far along the curve. So, 0% is going to be the start of the curve, 100% is going to be the last point on it, etcetera.

Position On Curve also has a few other inputs that may be worth noting. Follow is a boolean input that causes the camera to point in the direction that the curve is facing. If you uncheck this, you can supply a Look At BB that points the camera towards another chess piece while it is doing this loop. There is also a Bank parameter that must be used in conjunction with Follow, which causes the camera to tilt as it turns along the curve.

The above should be helpful for getting a sense of what the camera scripts do, and how you can use those building blocks to in your own camera scripts. Whenever you see a building block that you want to use, you can tap F1 on your keyboard to see the documentation entry for that BB. This will tell you what folder the BB is under so you can find it when you want.

**Creating a new scripted camera**

Let's create a new camera in a scene and have it do something interesting.

When you create a new camera, it automatically has the position of the current camera that you are using. You can use the movement and navigation tools to get the camera in the right position, and then click on the new camera button on the side of the screen. When you move around, it may be helpful to switch between different camera views, which you can do at the top of the 3D Layout pane.
I'm going to create a new camera here and rename it to "Vertigo cam", and set an initial condition on it, then create a script on it. I need to make sure that the script is not activated at scene start so that it can be triggered normally.

The first thing that needs to go in the camera script is Set As Active Camera. Because I like the effect, I am also adding a Vertigo BB to the camera. This is under Cameras → FX → Vertigo. This BB takes a few parameters, specifically, a focal point (which is necessary, but it does not need to be in the center of the field of view, I'll choose white_pawn_4), a number of frames the effect should last, and a distortion amount. This effect is another loop BB, so it has a loop input and output, so I am going to loop those around.

In order to make sure that this script is triggered, I am going to put it in the array of moves right after establishing shot1 Script. Now, if everything is alright, it should run on the second chess move.
08: Sound

Download the archive for this lab
This lesson covers how to work with sounds and music in Virtools. It is actually quite easy to use these, but there are several options for how to use them that gives you a great deal of control over their playback. The most significant of these is the use of 3d sounds. This lesson will also cover how to control a listener via 1st or 3rd person, and use that to navigate the world and hear the sounds in it.

**Acquiring sounds**

There are a variety of places that you can turn to for finding sounds. I'm not going to cover these, because they should be pretty easy to find for yourselves. To import sounds into Virtools, you must make a Data Resource folder, and place it and its RSC file in the same directory as your composition. Most resources, such as models, textures, and animations become embedded in the CMO, but sounds and videos do not. Thus, to make sure they play, you need to make sure you keep the CMO and the resources together whenever you move the files.

Virtools can recognize several different audio file types, such as wav, mp3, as well as others. When you open your resources, you can see the sounds and simply drag them into the scene to make them available for use. In the screenshots I have used the standard Virtools Resources folder.

**Playing sounds**

There are two categories of differences between sound types: Streamed sounds vs instance sounds, and 3d sounds vs background sounds. The first set of differences determines how sounds are played. Streamed sounds are played via the "Wave Player" BB, and instances, such as sound effects, are played via the "Play Sound Instance" BB. Streamed sounds are good for music or background sounds, whereas effects and UI feedback sounds are much more effective as sound instances.

The difference between 2d and 3d sound types lies in the space of the sound. 3D sounds are effective as
sounds that should be coming from objects. These can be either effects or ambient noises. Virtools can set a listener and determine how the listener distance affects volume, the panning of the sound, and also doppler effects according to the relative movement between the listener and the sound source. 2D sounds are much less complicated. These have no location or explicit listener, but they behave very predictably. 2D sounds can be used very effectively as interface sounds and background music.

The following schematic is the level script for sound01.cmo, and plays some 2d sounds. The script does the following: It uses a 2D picking to identify objects in the scene, and will play a different sound depending on which one was picked (by the Switch On Parameter BB). If a sound is played, it will wait for 1 second and enable picking again. The script also activates a looping ambient background noise.

Sounds can be played in response to messages, keyboard or mouse events, and basically anything that you can think of.

**Using 3d sounds**

To configure sounds so that they are treated as 3d is slightly tricky, but ultimately does not involve
much in the means of changes. For both instance and streamed sounds, you need to set the Type to be a "Point" instead of "Background" sound in the setup. This will give you many options for controlling the options for spatial playback. If you are playing a streamed positional sound, you can specify here the location where the sound is being emitted from. This can be an object or 3d frame. One thing you will wish to change are the minimum and maximum perception distances. I like to choose a large value for the max perception distance, since this will make sure the sound fades properly when you are far from it.

When you play streamed sounds, you can play them normally, if you specified a source within the setup. With instance sounds you can actually specify the source within the Play Sound Instance BB. Using this you can easily have the same sound being emitted from several objects simultaneously.

In order to actually have an object in the scene serve as the listener, you must use the "Set Listener" BB, which you can provide a 3d frame, or an object, or a camera. By default the active camera will serve as the listener. The code that uses this is in sound02.cmo, which has three ambient sounds being played from objects, and a instance sound that is played via clicking a 2d frame button. The listener is the ship in the middle which can be translated to hear the sounds differently.
Moving the listener

Now that we have a listener in the world, we can control it via keyboard commands. In sound03.cmo, the rocket ship is controlled via the arrow keys. The up key translates the ship along the -z axis, the down key translates the ship along the +z axis, both in relative space. This means that when the ship is rotated, movement will always be relative to the current orientation of the ship. The left key is mapped to rotate the ship a positive rotation (which is counter-clockwise), and the right key rotates the ship negatively, both about the y axis.

We can continue with this movement logic to permit the camera to follow the rocket as in sound04.cmo. This is done very easily. We can start by creating a new camera, and then giving it a script that will set it as the active camera, and then applying the "Keep At Constant Distance" and "Look At" constraint BBs. This permits us to cause our camera to follow the rocket wherever it goes. You must be careful about the parameters in the building blocks, though. The constant distance will need to be approximately 50 for the scene to work effectively. More advanced explicit camera controls can be enabled by making use of the other building blocks as well.
Another tool that can be used is the "Floor Slider" BB (under Collisions/Floors), which should be applied to the rocket script. In order for the floor to work, the floor objects must have a "Floor" attribute set, under the "Floor Manager" category. This prevents the rocket from leaving the floor boundary. An example of this is in sound05.cmo.

Finally, you can create collision detection by creating new group of objects that will be collidable, and then using the "Object Slider" BB, which creates a collision restraint on the specified group of objects. In order for this to work I have also added the "Object Keep On Floor V2" BB in sound06.cmo.

**Advanced sound**

This is fairly sophisticated stuff, but you can do quite a bit more by using parameters and parameter operations. You can alter pitch, volume, and panning explicitly via some given building blocks, and you can also get levels via using the "Get Sound Spectrum" BB. This can allow you to do some neat things. The file sound07.cmo creates an equalizer using several different objects and applying transformations based on the sound spectrum. It does this by using several different parameter operations to get the position of the object, and set its y value to be the sound spectrum result.
multiplied by a specified value.

To create a parameter operation, right click in the schematic and select "Add Parameter Operation", or press ALT-P. Parameter operations take one or two inputs, and have a single output. You can identify a specific parameter operation by selecting and locking its input and output types, or its name. Usually you can perform some useful and intricate operations by using them.

09: Advanced Scripting

Download the archive for this lab

I want to cover a few final topics in scripting before you turn to work exclusively on your projects. Scripting in Virtools has always been tricky for students, and it is my hope here to present a few ways that will help you keep on track of scripts for your own projects.

Loading and Saving

Most scripts can be saved and loaded via right clicking on a script or object name in the Level Manager, and specifying save or load script. This will prompt you to load or save a *.nms file, which can be reused in scenes and shared easily. This is a good way for you to keep your code packaged and modularized so that your projects go smoothly. Later on we shall see that Behavior Graphs and VSL scripts can be saved in the same way, and then dragged and dropped into a scene.

On Parameters

You have seen local parameters all over the place, even if you haven't recognized them as such. A local
parameter has three properties: *name*, *type*, and *value*. When you create a building block, it automatically creates local parameters for each parameter input that it has. You also make your own local parameters, and connect them to other inputs, and even use them as things Virtools calls *shortcuts*. The important part of parameters are the types and values, and we will explore those presently.

If you right click in the schematic of a script, you can select an option to "Add Local Parameter". When you do this, you will get one of those gray rectangles that we have come to know and love. If you double click on it, you can access an Edit Parameter box that tells you the name, type, and value of the parameter. This one is a float, but if you click the drop down for the parameter type, you can get a LOT of different types of parameters.

One of the other more useful types is Vector, which is a 3 dimensional vector indicating a direction or a point in space.

If you select a parameter and press the spacebar, you can toggle between the way it appears. Pressing the spacebar once will change the gray rectangle into a square and show you the name of the parameter. Pressing it a second time will show you its value, and a third time will show you the name and value. You can also right click on the box and select "Change Parameter Display". Using the value display is extremely useful for doing debugging and making your scripts easy to read. When you have Trace mode turned on, you can see the value of the parameter as it changes.

**Advanced work with Building Blocks**

To start with, I want to make sure everyone knows that if you hold down CTRL, and double click the mouse, you get a search box that allows you to find any building block by name. This is very useful if you know vaguely what you want, but aren't sure where the building block is located. A good use of this is to find the "Switch On Parameter" BB, which is oddly located in Logics/Streaming.
Switch On Parameter is one of several building blocks that allows you to perform some rather detailed customization with it. The purpose of this particular BB is to take an input parameter and redirect control depending on what parameter is specified. This is basically like doing switch/case or an extended if/else statement in regular programming. You can test numerical values, but you can also test versus other sorts of values, such as 3d Objects, animations, sounds, etcetera.

The problem with the building block when you create it is that it is automatically set to use the float type, and you need to do work in order to change the type to the one you need. You can do this by double-clicking on the parameter input arrow on top of the building block, as shown here:

From this point, you can change the input type to whatever you please, and specify the input parameters accordingly.

There is one more thing that you can do with this particular building block, and that is to use the Construct feature. With this and several other building blocks you can construct behavior outputs that correspond to automatically formed parameter inputs. This function is available on building blocks that have a small "V" in the lower left corner of the building block. If you right click on the "Switch On Parameter" BB, you can select Construct → Behavior Output, which creates an additional output, and a corresponding input:
Many of the controls provided by the *Construct* feature are repeated in other Virtools structures, such as Behavior Graphs, as well as VSL. In general, there are two axes that operate on building blocks. The horizontal axis denotes behavior, or flow, while the vertical axis denotes parameters, or information. This trend carries through with all of the structures that work in the schematic. In the schematic you are working with flows of information and behavior, and you need to operate on both in order to elicit interesting and complex scripts in Virtools.

![Diagram of schematic with axes labeled Behavior and Information]

**Behavior Graphs**

This was mentioned in the previous lesson, but bears mentioning again. A *Behavior Graph* is a subsection of Virtools script that is grouped together. You can use behavior graphs to simplify a repetitive and complex tasks within the schematic. To draw a behavior graph, right click in the schematic and select "Draw Behavior Graph". Then you can click and drag the mouse to define a region that makes up the behavior graph. This region serves as a miniature schematic, and where you can treat a subsection of a script as a single building block.

You can construct behavior and parameter inputs and outputs, which allows the graph to interact with the rest of the script in any way you wish. Also, you can add building blocks and parameter operations, allowing you to define the behavior graph to do any complex task that can be done ordinarily. When you construct your input and output parameters, you can specify the names and types thereof, which allows you to define the graph as essentially your own custom building block.

One of the chief useful things about a Behavior Graph is that you can copy and paste it, and then make changes to one and update those changes across all instances in a scene. Additionally, you can save and load and update from a saved file. This allows you to store a behavior graph as a single component that can be quickly dragged into your scene.

**Breakpoints**

No lesson in advanced programming is complete without a section on Breakpoints. If you have done programming or debugging in any sort of standard development environment, you probably know what breakpoints are, but that may not be all of you so I'll explain here.

When you are writing a script in Virtools that has a lot of connections and it is hard to tell what is going on, it is useful to have something that will pause playback when a certain condition is reached.
Breakpoints do just that. If you right-click on a connection in the schematic, you will find an option to "Add Breakpoint". This allows you to create a breakpoint on that connection, which will freeze execution whenever that connection is activated. To remove the breakpoint, right click on the connection again and select "Remove Breakpoint".

![Breakpoint Example]

Breakpoints are the most useful when you want to see whether something is not happening when it should, or when something is being called when it shouldn't be. The use of breakpoints will allow you to diagnose a lot of commonsense problems that can emerge in Virtools.

When the execution stops, you can observe the scene and play it back frame by frame in order to identify what changes are taking place. If you change the parameter display, you can watch as parameters change their values over each tick of the building blocks.

**VSL**

The final bit that I want to cover in detail is VSL, for the Virtools Scripting Language. VSL is basically a hybridization of C/C++ with Virtools logic. This exposes to us a layer of Virtools underneath the schematic. Using VSL, it is possible to write your own building blocks using procedural code, instead of working from other building blocks. This way, it is much more easy to perform math operations and to do loops and other such control structures. VSL is also compiled, so it operates on a much faster level than regular Virtools behaviors, so if you get a code related performance slowdown (as opposed to an asset based one), VSL may improve things substantially.

To create a VSL script, locate VSL in the building blocks menu, and drag "Run VSL" into your schematic. When you create it, it looks like a simple building block, but double click on it, and you get something wonderful:

![VSL Editor]

Actually, let's take a closer look, that's a little hard to see while zoomed out:
Why, it's a regular "void main() {}" block, incredible!
Virtools allows us to do traditional procedural code, and structure this as though it were a building
block. In order to do this, we can specify (akin to the Construct tool), different parameter and
behavioral inputs and outputs. The behavioral inputs and outputs are represented via boolean values, so
depending from which the VSL script was entered from, the appropriate bIn will be true, and the rest
will be false. The controls for these are on the right side of the screen:

If you right click on the pIn section, you can select "add pIn" to add a parameter input, and specify its
name and type. Some of the types available here are different from the regular types available in
Virtools script. The reason for this is because VSL code, like most C-style languages, can not give
something a name that begins with a numeral, such as "3D Object". Instead, VSL denotes that
particular type as "Entity3D". Most of the others, however, such as "Animation", "Curve", "Camera",
etcetera, do not need any modification other than removing spaces.

Virtools scripts also must be compiled in order to work in a scene. Usually, you must do this yourself,
by right clicking on the script in the left tab, and selecting Compile, or clicking the "Compile All"
button. I have had scripts automatically compile on occasion, but this is unreliable at best, so I suggest
that you compile manually before testing a script.

VSL has very effective syntax completion, so if you have used any relatively recent IDE for
programming, you will probably notice that if you begin typing a name (such as the name of a pIn), it
will suggest the name, and if you press ENTER, it will fill the name in for you. If you type a ".", as
VSL is slightly object-oriented, it will suggest methods that you can call on that object.

Attached in the lab files is a cmo which has a VSL script that creates a strobe light. This should be a
good example for how to use and extend your own VSL scripts.