

Tricks, Trinkets, Whatnot

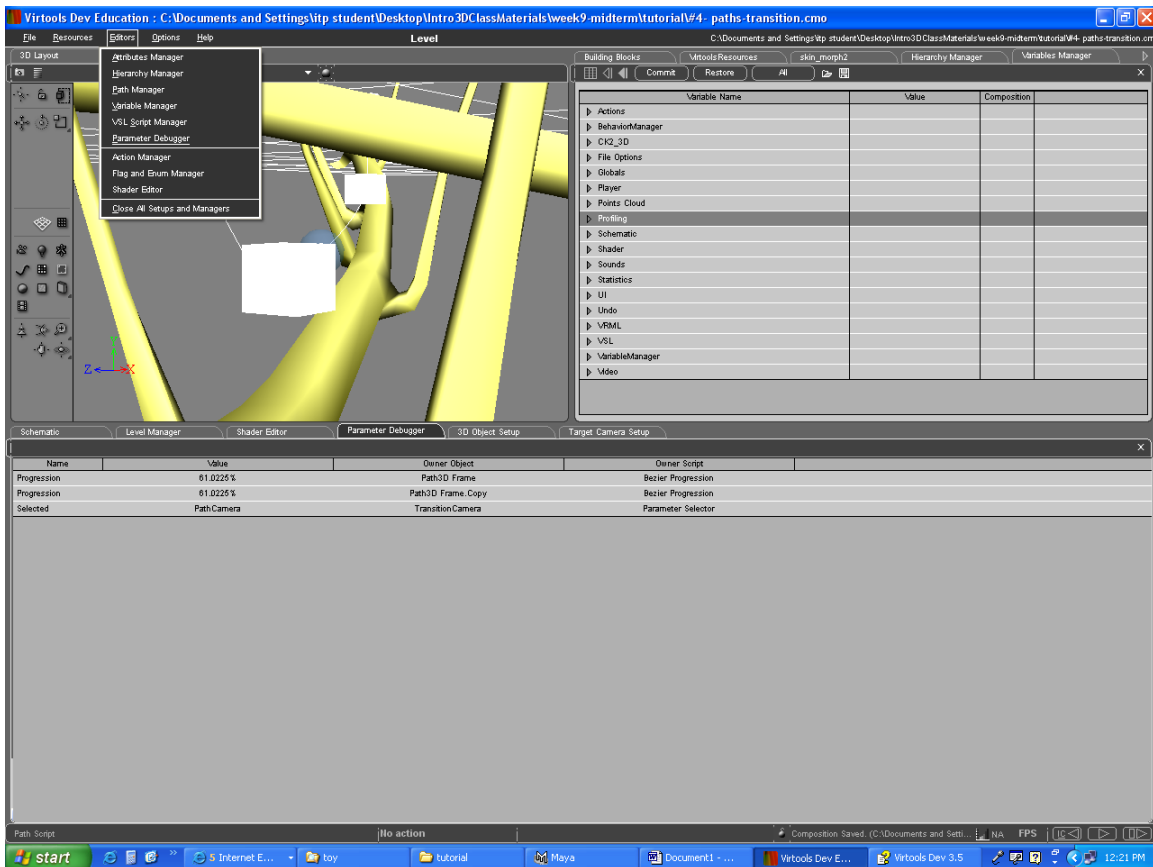
At this point I think it's worthwhile to cover some miscellany that may be helpful in both increasing your Virtools productivity and adding some "points of delight" that while they may seem insignificant, can end up adding a great deal of polish to your compositions.

Virtools DDT [Devastating Debugging Techniques]

As your scenes begin to get more complicated, it can become difficult to keep them perfectly organized. Occasionally building blocks will begin to interact in initially mysterious ways that are difficult to diagnose from superficial inspection. Like any full service development environment, Virtools provides a variety of tools to help you debug your recalcitrant compositions.

Parameter debugger

Perhaps the most important debugging tool you can use is the parameter debugger. You can access this delightful tool from the Editors -> Parameter Debugger menu item:



To add parameters to track, simply control select them in your schematic and the paste them into the parameter debugger window. You can select parameters from many different scripts at one time.

Be sure you hit “trace” in the schematic window so the parameter values are updated in real time. Then play your composition and you should see the values change as your various scripts execute.

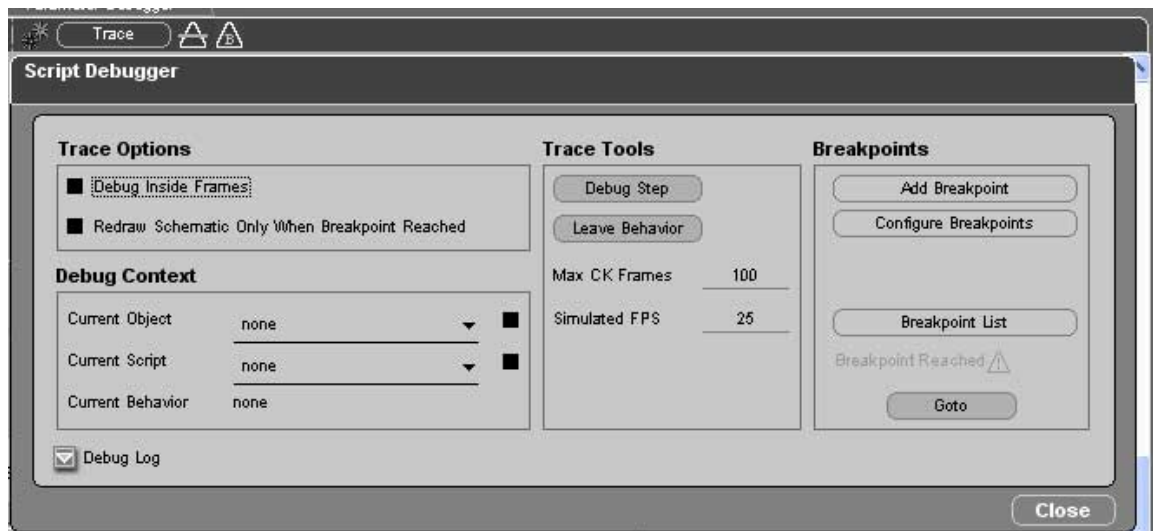
This tool is essential for determining that the correct data flows into your BB’s at appropriate times.

Also, please note that it can be very helpful to pause your scene and use the “Step One Frame” button (to the right of the play button) when you are examining your data in the Parameter Debugger.

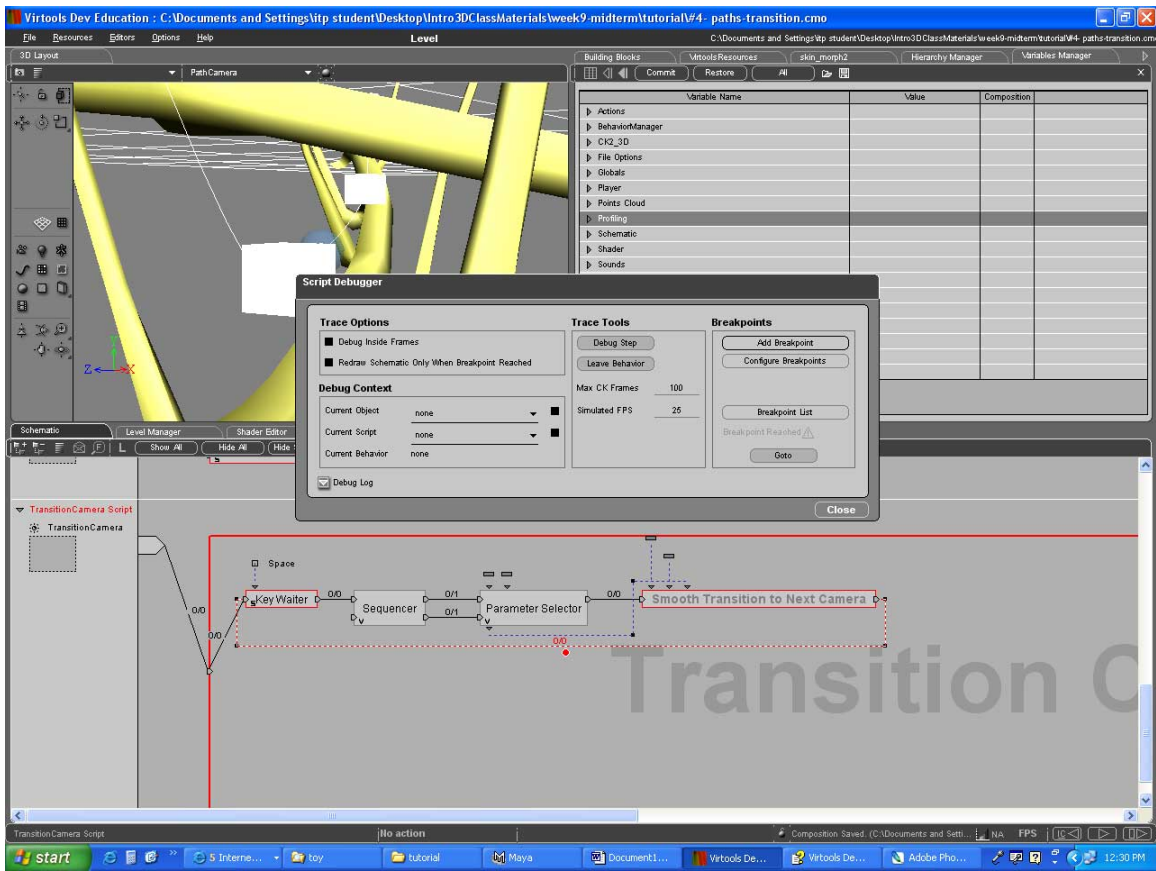
Script Debugger

Frequently your data is changing so quickly that it is hard to see exactly when crucial transitions are occurring. We already discussed increasing the link delay on Blinks so that you see a countdown as a link is activated. However, due to the fact that this does not actually stop your program and can indeed disrupt its execution in some cases, it is not always the preferable method.

As with any programming task, it is essential to be able to stop your program at specific points to inspect exactly what is going on. To do this in Virtools, just like any other environment, you need to set break points. Virtools offers a handy little button (with an icon of 2 little gears on it) directly to the left of the “trace” button in your schematic view that allows you to do this.



If you click the “Add Breakpoint” button, your cursor will turn into a weird little pointer that allows you to click on Blinks and therefore set a breakpoint which will show up as a red dot. Here we have selected the link that gets executed after you have pressed the space bar in the transition cam demo.



Now, every time your script executes this Blink, the program will stop, and you can inspect Object Setups, or variable values in your script debugger. This can be unbelievably helpful in determining what's going on with your program.

N.B. Be sure the "Trace" function is turned on, or breakpoints will not be taken into account by the player.

Another fundamental debugging ability is to break only when certain conditions are reached, such as when a specific Blink is getting executed every frame, and you only want to break when a certain variable reaches a certain condition that is supposed to effect some transition that doesn't appear to be happening.

To put a condition on one of your breakpoints, open the script debugger and hit the "configure breakpoints" button. You'll see the breakpoints configuration dialog pop up. Hit "Add Condition" and then "Pick Schematic Parameter," then select a Pout or Pin from one of your scripts (it doesn't have to be in the same script as the breakpoint). Here I selected the "progression" Pout from one of the sphere Bezier BB's. I then double clicked the "condition" to change it from "=" to ">" and I changed the progression value to "50%." As currently configured, this breakpoint will now break only when the sphere is over halfway done with its progression. Try it and see.

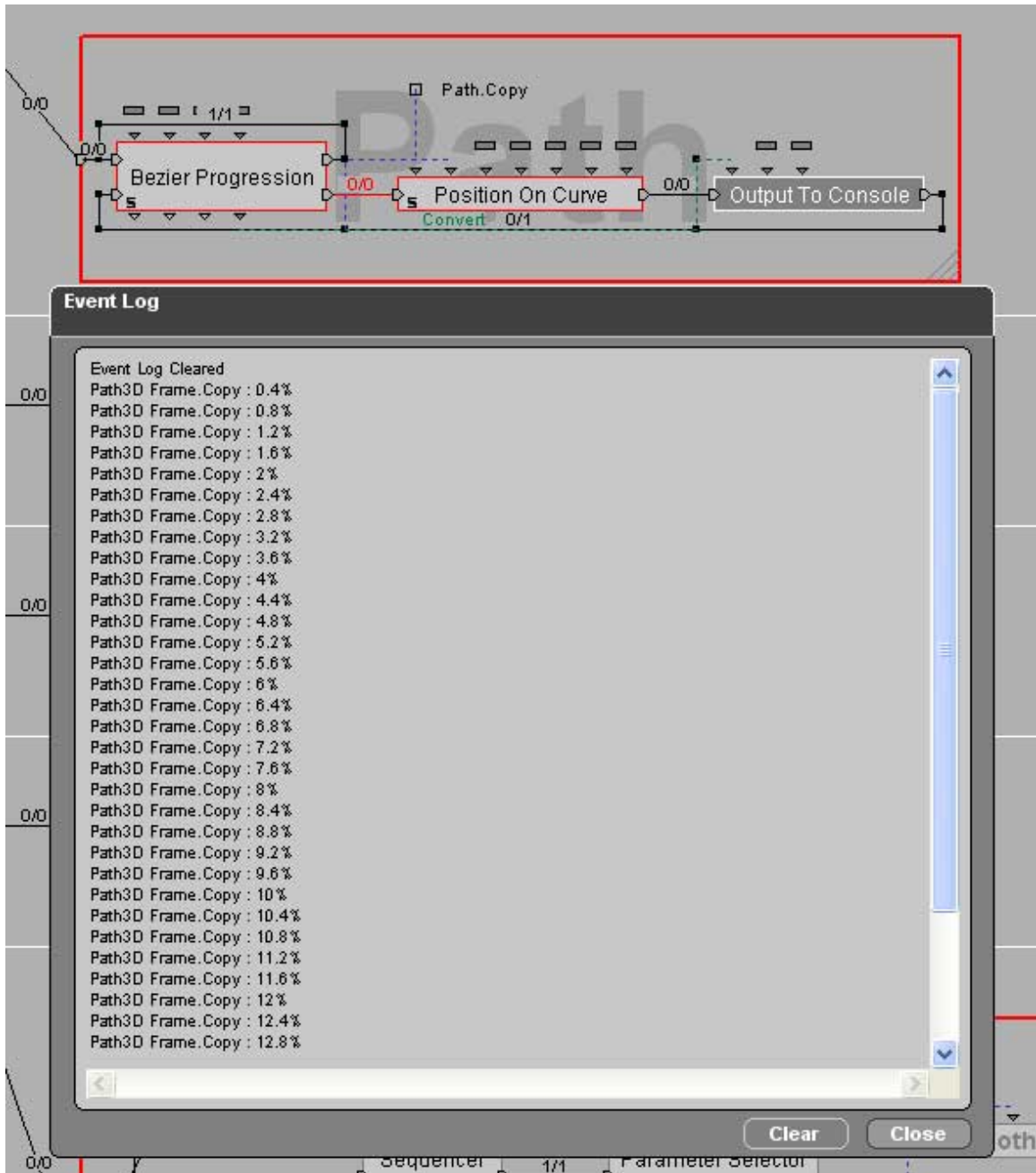
Once you have reached a break point you can literally run through the Virtools behavioral processing loop one step at a time by checking the "Debug Inside Frames" check box and successively hitting the Debug Step Button. You can tell

where the program is by monitoring the "debug context" text boxes and watching your scripts' various behaviors light up.

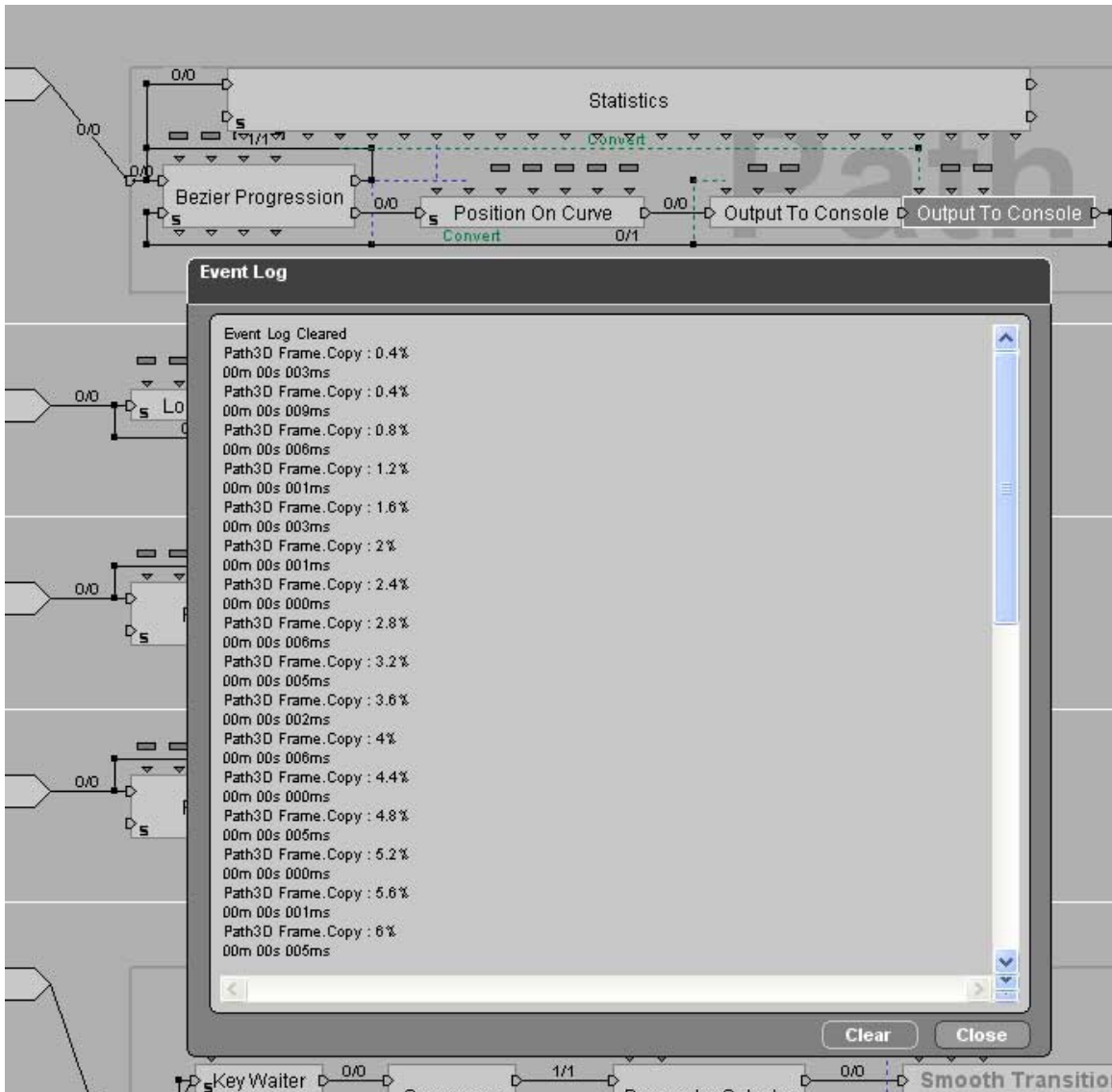
Output to Console

When you have extremely complex behavior configurations that you're having problems tracking even in the debugger, sometimes it can be very helpful to create a written log of exactly what's going on in a program under certain situations.

The Logics -> Streaming -> Output to Console BB is quite instrumental in this regard. All it does is print strings that you can configure into the Virtools debugging console. The console is accessible via the little "unlocked lock" or "bomb" icon down to the left of the IC and Play buttons. Below is an example of an Output to Console BB integrated with one of the sphere's motion scripts. I'm having it output the Bezier BB's progression parameter.



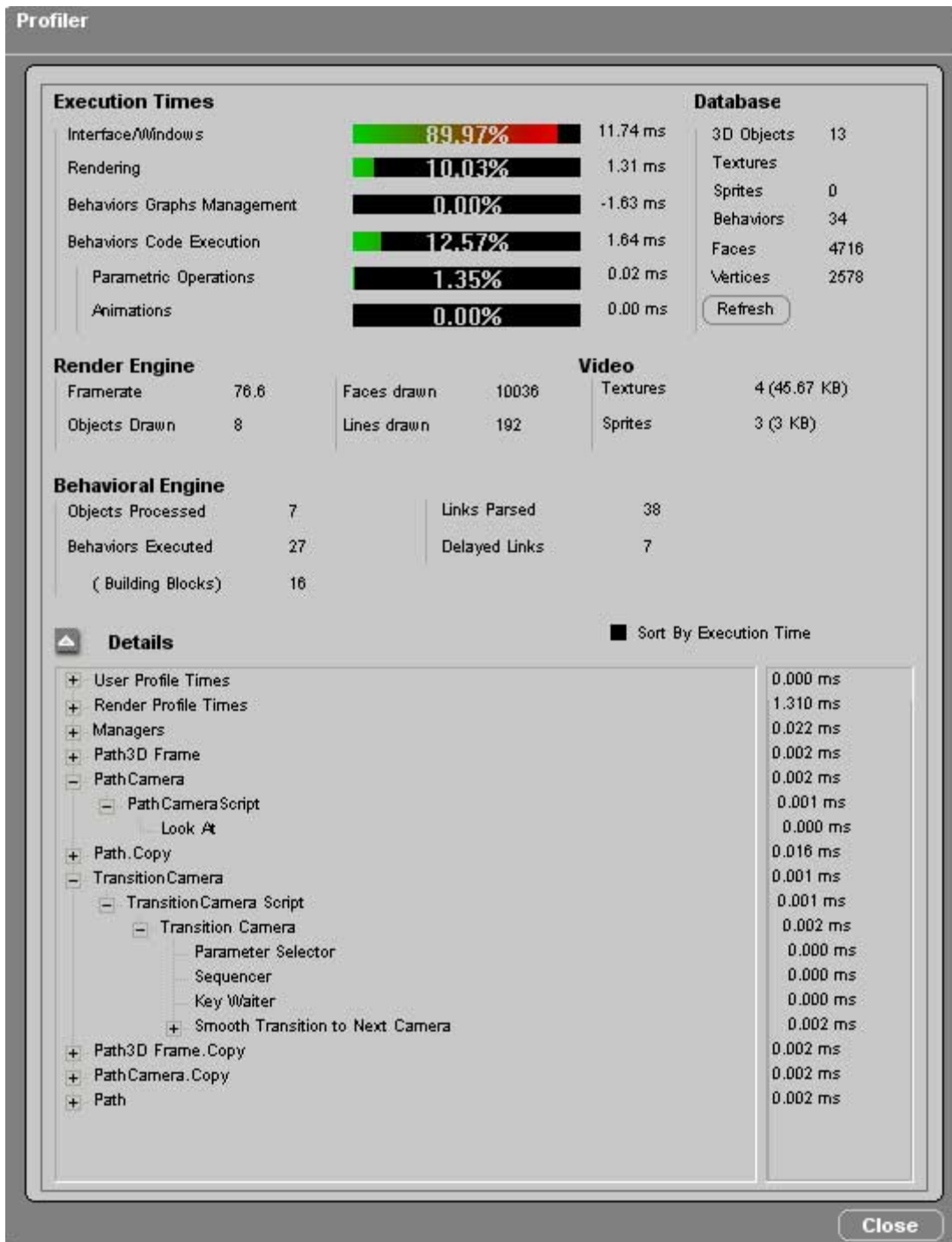
If you add the Optimizations -> System -> Statistics BB to another Output to Console BB, you can include a great deal of system information about your scene's behavior as well. Here I'm recording the time it's taken to render the scene on every frame:



Please note that the statistics BB can eat up an enormous amount of processing time & cause your .cmo's to run very slowly, so you should always disconnect them once you've solved your specific problem.

Profiler

A potentially simpler way to get a good picture of how your composition is spending its time (and perhaps why you're getting a suboptimal frame rate) is to use Virtools' profiler tool. This is accessible via the graph icon immediately to the left of the frame rate readout next to the "play" button. As depicted below, it shows the amount of processing time your composition is using in every category of function down to individual behavior in your objects' scripts.



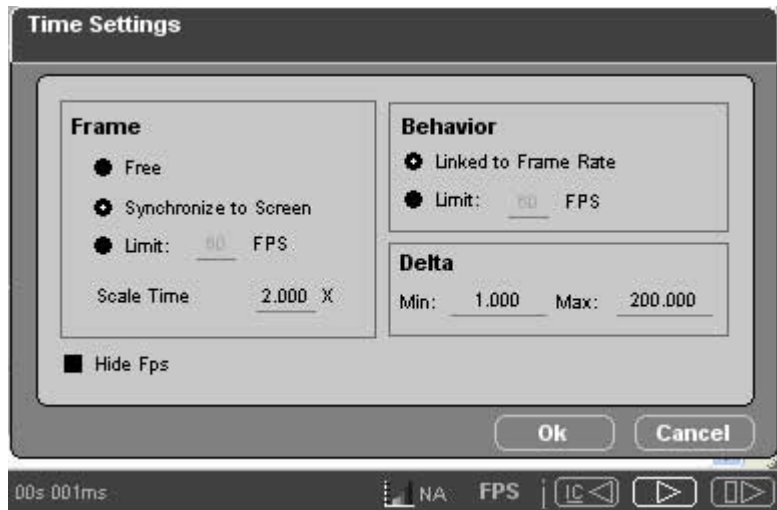
Watching these numbers change as you interact with your .cmo can go a long way toward explaining some types of strange behavior not caused by explicit programming errors, and can greatly aid in optimizing your scene's performance.

Time Settings

While we're on the subject of performance, as many of you have already noticed, a given scene's frame rate is highly correlated to the hardware specifications of the

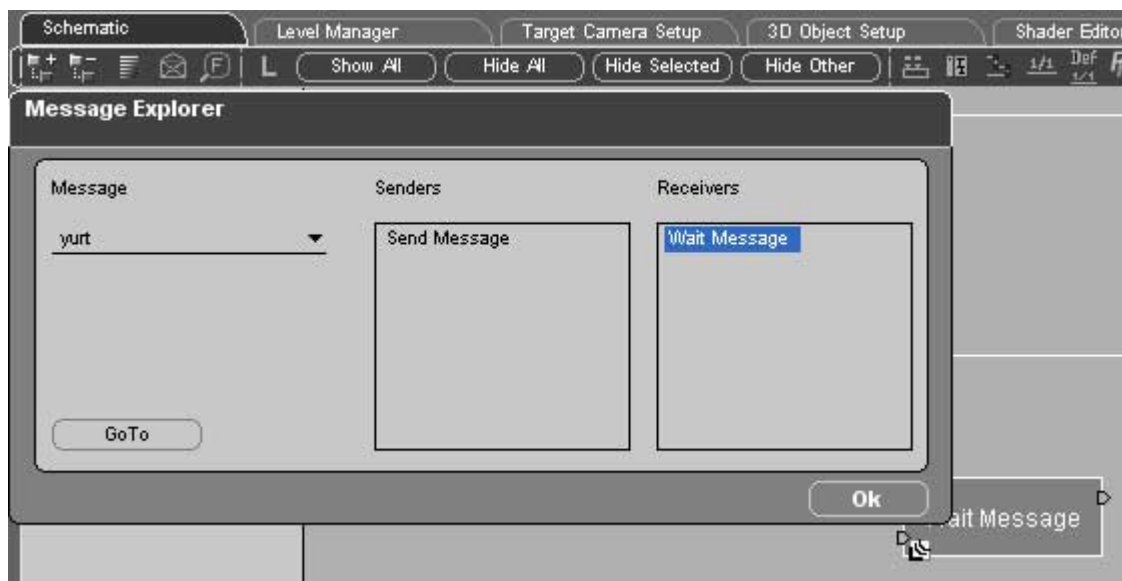
machine on which it is running. To make your compositions' performance more consistent across various platforms you can use the Time Settings dialog to set a frame rate limit geared to a "minimal" system, and then proceed with the expectation that your scene will perform similarly on another system as long as it has "as good or better" specs.

You can bring up this dialog by right clicking the play button.



Message Explorer

A final tool that can be helpful in untangling complex message chains is the Message Explorer. This is found under the envelope icon just underneath your schematic tab. It essentially allows you to select any message in your scene, and see which behaviors are sending and receiving it. You can locate these in your scripts by selecting one, and then hitting the GoTo button.



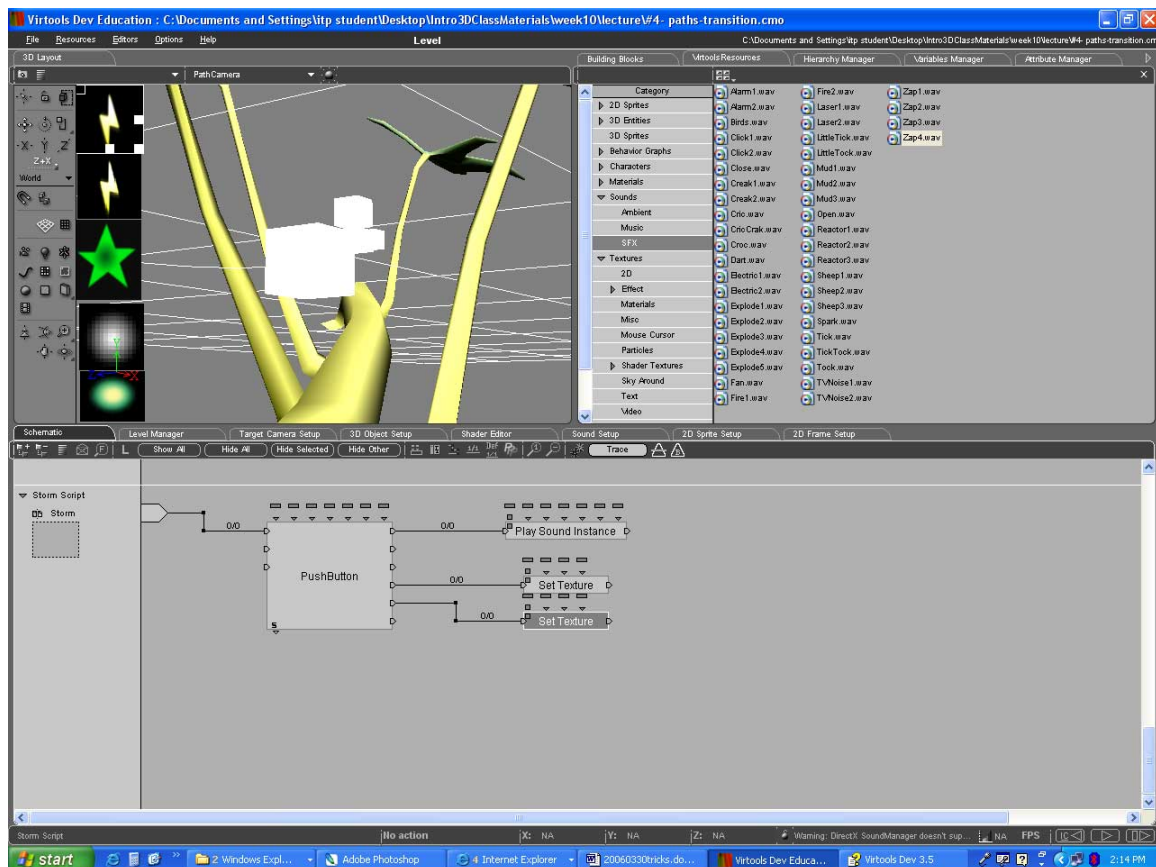
Now that we all have the ability to get our scenes running like tweaking cheetahs, let's look at a few "points of delight" techniques with which you can decorate your scenes.

2D UI

As many of you have discovered, Virtools provides some fairly robust tools to create 2D interface elements which allow you to implement significant functionality for your scenes. The simplest way to see this in action is to grab a .jpg texture from your resource file and control-drag it into the 3d layout. This will create a 2D Sprite with the specified texture in the current view.

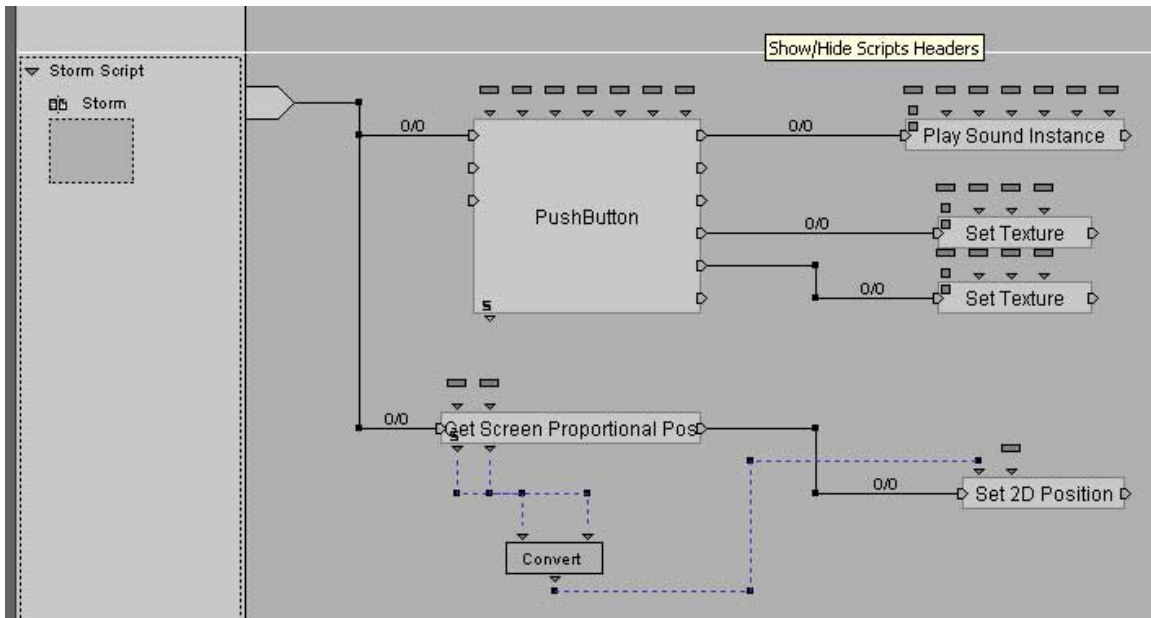
Note you can control-shift-drag to create a *2D Frame* with the attached texture. You can shift-drag to attach the texture to a *3d Sprite*. For the various advantages and demerits of these creatures, please refer to Virtools 2D interface whitepaper.

The two main functional BB's for Sprites are the PushButton BB and the Drag and drop BB. For example, put a script on one of your 2D frames. Add the Push Button BB, and connect a Sounds -> Basic -> Play Sound Instance BB to the pressed output. Play the scene, and you should hear your sound play when you hit the button. You can also change the frame's texture with the Materials / Textures -> Basic -> Set Texture building block on "Enter & Exit Button" outputs to create "rollover" effects.

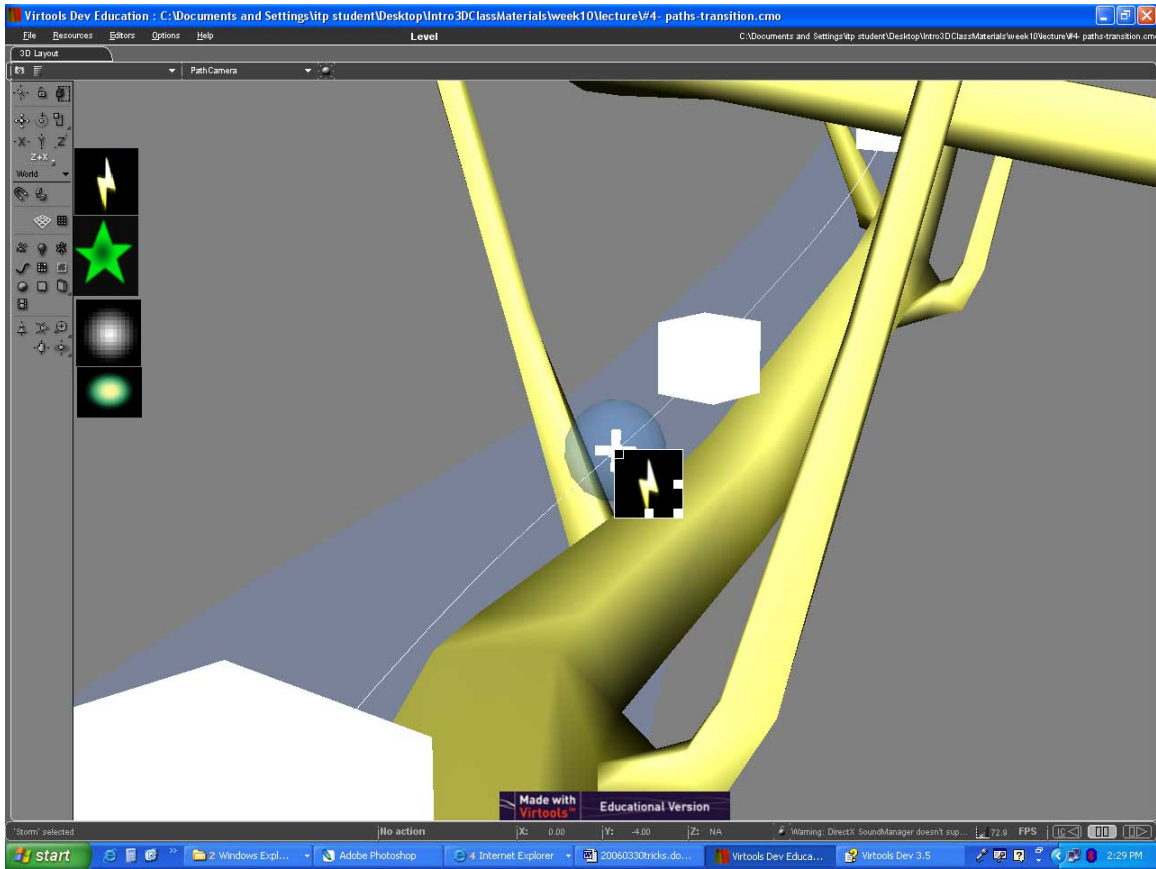


It is frequently helpful if you have multiple push buttons to have them simple send an `x_button_pressed` message to a central message handler to facilitate functionality or mode changes in your scene.

An important thing to note about 2D interface layouts is that they tend to be situated with absolute coordinates, and so changes to the view size can disrupt your layouts unless they are left -top justified. If your layout needs to be proportional to the screen size, you'll need to work with the Interface -> Screen -> Get Screen Proportional Position BB and the Visuals -> 2D -> Set 2D Position BB. Like so:



This will now set the frame's position to roughly the middle of the screen, no matter how large it is.

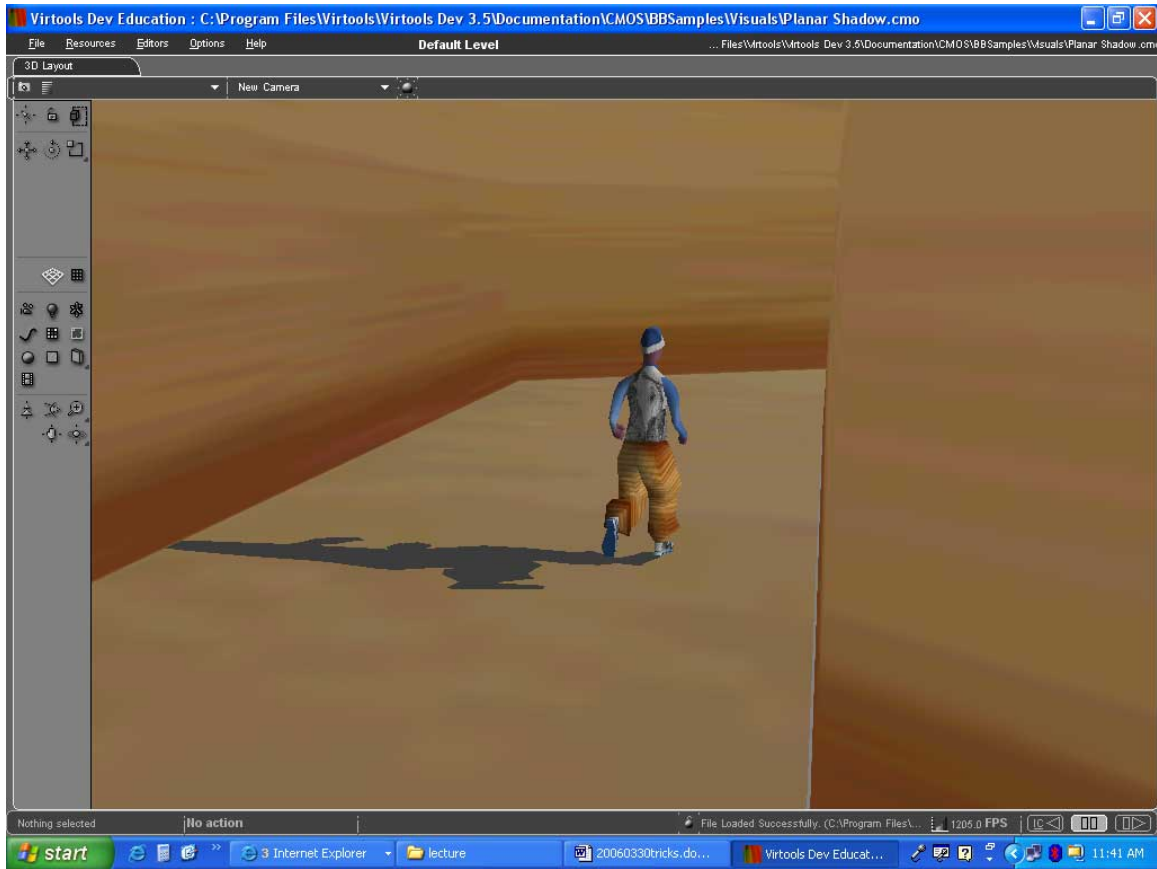


Shadows

Shadows can be a great way to add richness to your scenes, especially your spooky graveyard scenes. Upon initial inspection, shadows can be a slightly bewildering area in Virtools, since they offer four ways of dealing with them. This is an artifact of the days when, given that it can be a fairly computationally intensive operation, shadow casting on older systems could cause rather severe performance problems. Thus Virtools offers 4 different ways to cast shadows of more or less increasing complexity:

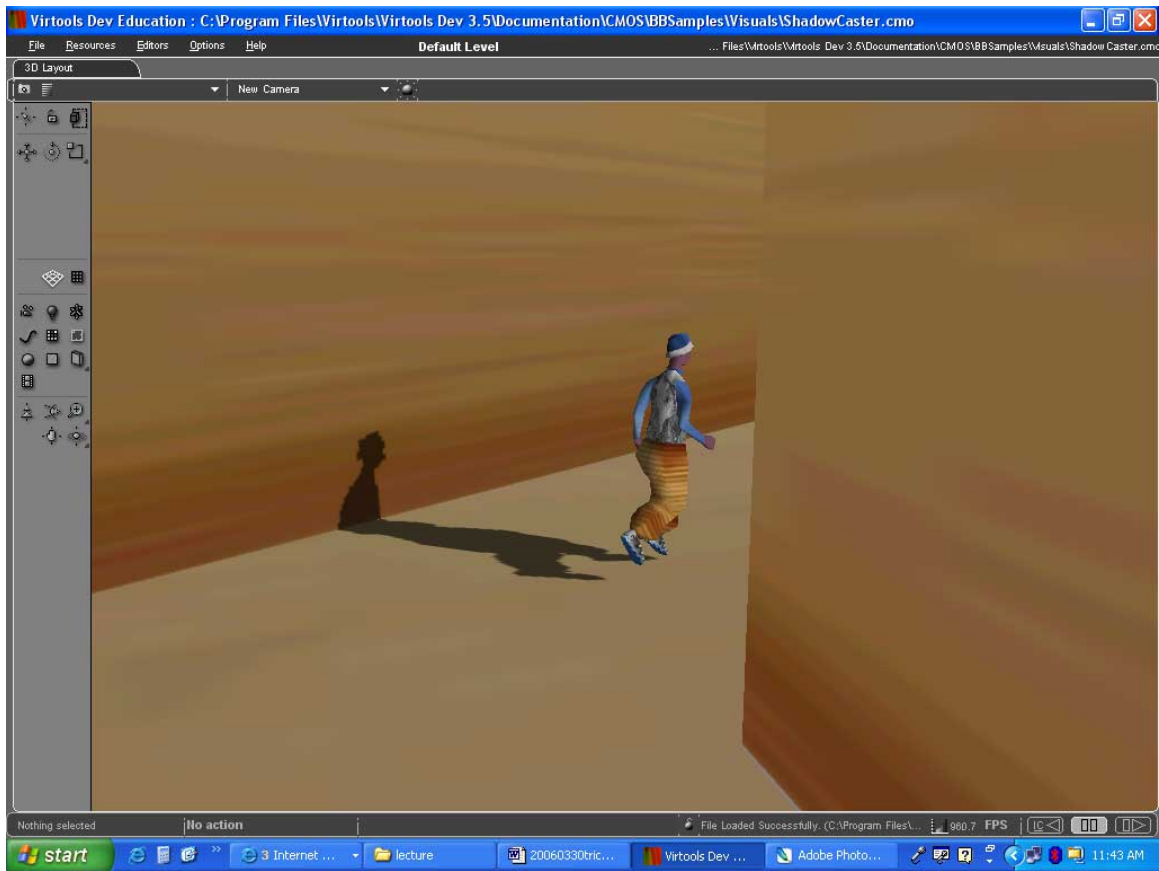
Simple Shadow – basically has a shadow texture follow a character or object around. Do not use this method unless you absolutely have to for performance reasons.

Planar Shadow – A low resource, but reasonably realistic way to calculate shadows. As depicted below, a good shadow appears on the floor, but the head is cut off where it should continue on the wall. Essentially you put this BB on surfaces that should take shadows (the floor in this case), and then add the VisualsFx->Planar Shadow Object attribute to your character.



Shadow Caster – This is a delightful, high accuracy method of producing shadows. You add the “Shadow Caster” behavior to the entity casting the shadow, in this case your character, and then add the “VisualsFX -> Shadow Caster Receiver” attribute to surfaces (ground, walls) on which you’d like your shadow to show up. As depicted below, you get the character’s whole shadow. Note that running multiple instances of this BB can be very processor intensive. From the Virtools documentation:

The Shadow Caster building block may be used when you need accurate shadow rendering. To create the shadow, a rendering of the object casting the shadow is created in a texture. A dedicated camera is placed at the light position, pointing at the object that will cast the shadow, then this camera renders the object and all its hierarchy to a texture. This texture is used by a channel applied to the object receiving the shadow; therefore, this building block requires more calculation time than previous methods.



Shadow Stencil – this is another high-accuracy shadow rendering technique. The main advantage of this method is that the shadow of the object will cast on the object itself. However, your target system need to have a stencil buffer, and this method can result in the generation of unwanted shadows.

Virtools has a nice “Lighting, Shadows and Reflection Whitepaper” in their online help.