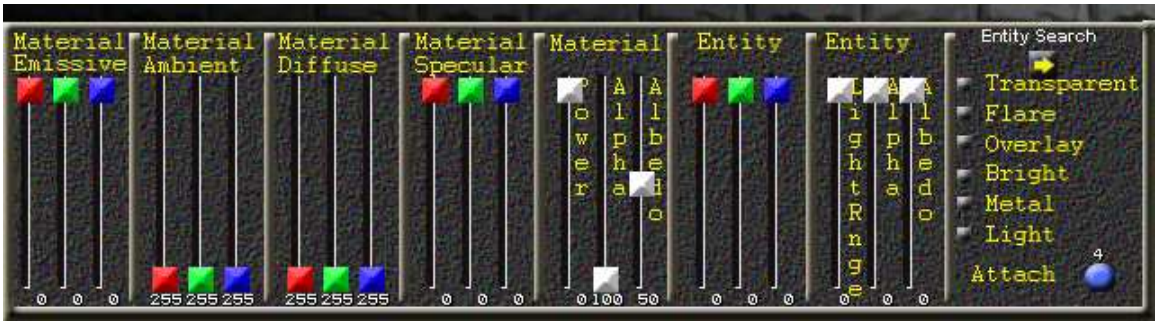
Light Mixer C-Script Version 1
For A6 Game Studio
Version 6.20

By George Schneider



Welcome brave programmer. If you haven't already done it, you might want to refer back to matTut1, the tutorial that describes how to use the Light Mixer Control Panel to study A6 lighting effects.

In this tutorial we are going to look at how to program the Light Mixer Control Panel in C-Script. It is actually a fairly straight forward program, except that it uses a very important programming concept called pointers. In some earlier versions of 3DGS they were called synonyms, but they are now being referred to by the more common computer term of pointers.

Pointers are somewhat hard to grasp, so I've written an introductory section on pointers and how they work. If you are already an advanced programmer and understand pointers, then skip right down to section 2 and see how Light Mixer is put together. If you are a programming wizard, and prefer reading code more than manuals, then you can skip this whole tutorial, and just open up lmx.wdl and and look through it. It has plenty of comments to help explain what is going on in the code.

I hope you find this useful in your own game development…

giorig3@mchsi.com

Section 1 - The nuts and bolts of using ‘Pointers’ in Light Mixer

Well, I am glad someone decided to look at this section.

I am going to take you back to the dark ages, when men were men, and programmers wrote in assembly language. We used to worry a lot about memory back then, especially where things were in it.

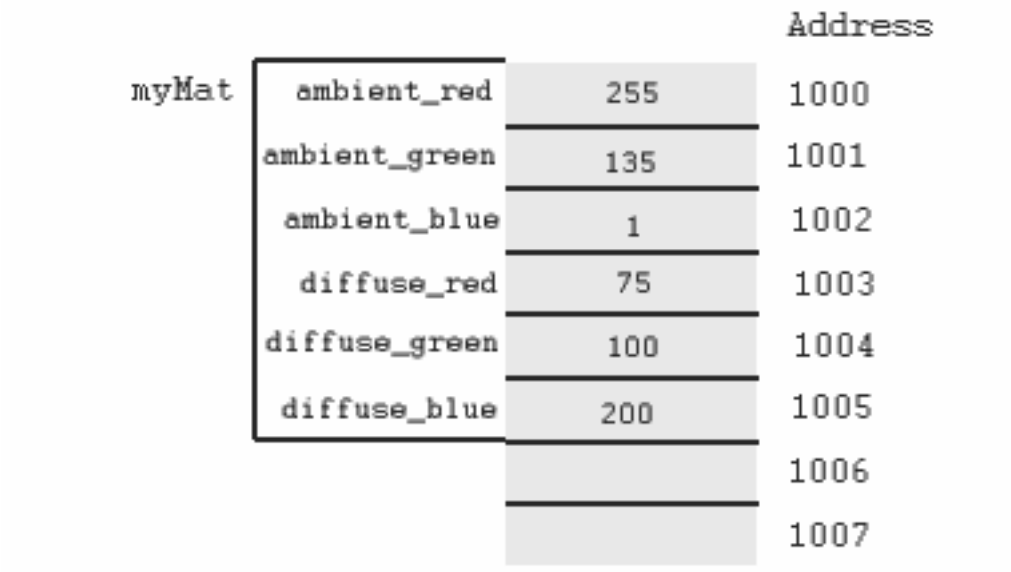
Let’s pretend we need to create a material structure in memory called myMat, that has 6 variables assigned to it, ambient_red, ambient_green, ambient_blue, diffuse_red, diffuse_green, and diffuse_blue.

(NOTE: There are more than 6 variables available for a REAL material structure in A6, and who knows what else. This is just fictional example of how pointers work.)

Let’s also assume each variable occupies 1 word (8 bytes, 32 bits) of memory. Let’s define our structure like this:

```
Material MyMat
{
      ambient_red = 255;
      ambient_green = 135;
      ambient_blue = 1;
      diffuse_red = 75;
      diffuse_green = 100;
      diffuse_blue = 200;
}
```

So let’s look at how the computer would see this:



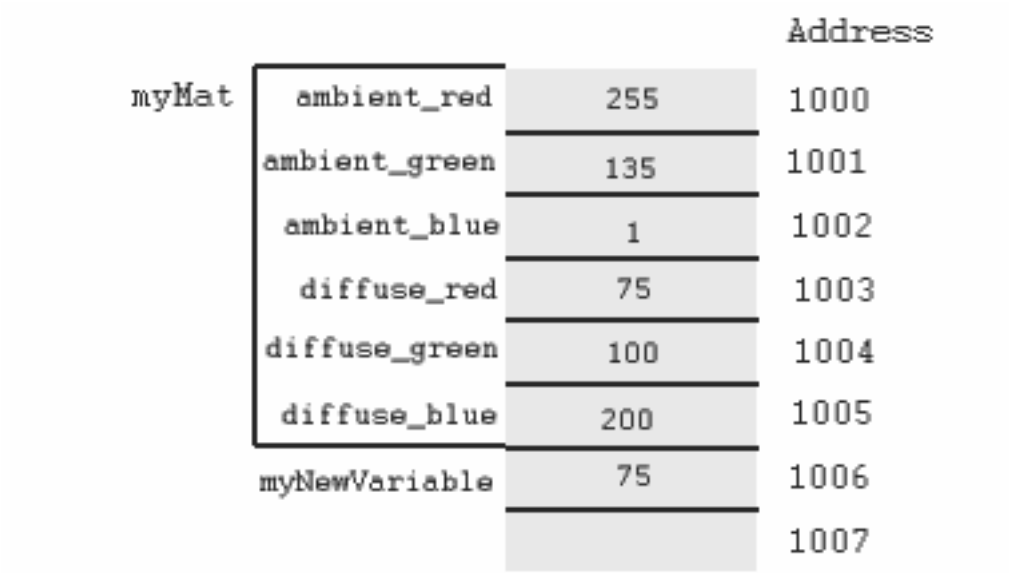
Here we have a Material structure in memory. Each line represents a word of memory. The number to the right is the word position in memory. The computer has assigned the locations for us. myMat starts at word 1000 and goes for 6 words, through word 1005. The text is simply a label (variable name) for us humans to be able to reference. By the time you have compiled your program, the computer no longer cares about the variable names, it

is strictly working with the word addresses and numbers. The numbers in the shaded area represents the value that we assigned to the variables as they are stored in memory.

So if we want to get the value of the variable diffuse_red, and put it in another variable, we would do something like this:

```
Var myNewVariable;
myNewVariable = myMat.diffuse_red;
```

And how would this look in memory? Like this:



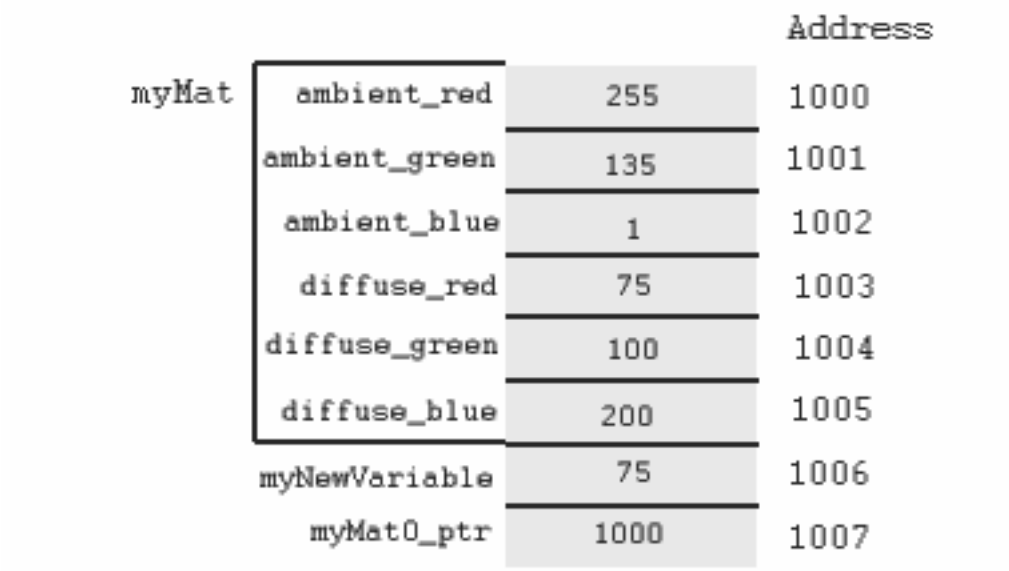
The computer assigned another word of memory (1006) to the variable myNewVariable, and then moved the contents of word 1003 to the word 1006. Ok, that seems simple enough.

So now let's declare a pointer. In C-Script we declare a material pointer like this:

```
material* myMat0_ptr = myMat;
```

This says to the computer, create a material pointer called myMat0_ptr and have it point to myMat.

Based on the above statement, memory would now look like this.



Again the computer created a new variable (myMat0_ptr), assigned it to memory location 1007, and loaded it with the value 1000. Where did it get the value 1000 for myMat0_ptr? Let's break down the statement:

```
material* myMat0_ptr = myMat;
```

material*	says we are declaring a material pointer, not a material structure.
myMat0_ptr	defines the variable name of our pointer
= myMat	says load our pointer with the ADDRESS of myMat

When we load a pointer with a value, we are telling the computer that we want the word address (i.e. 1000) of the variable loaded into our pointer, not the variable's value (i.e. 255). The pointer now POINTS to the same address as myMat. Why would we want to do this? We do this because it gives us a GENERIC way of referencing myMat. We can now write a routine that can reference myMat or any other material structure. This is a very important concept we are going to use a lot in the Light Mixer code.

Let’s write a little routine that does something to myMat structure. Its job will be to switch ambient_green with diffuse green and then increase the value of ambient_blue by 50. Let’s do it without pointers first, and then with pointers. In this example I’ve commented each line.

```
Function swapMaterial ()
{
 var local1;  // declare a local variable
 local1 = myMat.ambient_green; // save ambient_green in local1
 myMat.ambient_green = myMat.diffuse_green; // move diffuse_green to ambient_green
 myMat.diffuse_green = local1;      // move our saved ambient_green to diffuse_green
 myMat.ambient_blue += 50;    // increase ambient_blue by 50;
}
.
.
.
swapmaterial (); // call our routine to swap myMat material.
```

Ok easy enough right. Based on the picture above, **myMat.ambient_green would become 100, myMat.diffuse_green would become 135**, and **myMat.ambient_blue would become 51**. The contents of local1 are not relevant because it is a temporary local variable and goes out of scope when the function exits.

But what if we decide we have another material that we want to do the same thing to. Well using this style of coding we would have to write another function to do it, because our function specifically references myMat. No big deal right? What if this routine was hundreds of lines long. What if we had 100 material structures that we want to change in this way. Get the idea.

A pointer solves this problem for us nicely. Here is how we could code this routine using pointers. We need to define another pointer for our generic swap routine.

```
Material* genMat_ptr; // declare a new material pointer
```

Then we use the new pointer to reference out fields.

```
Function swapMaterial()
{
 if (genMat_ptr == null)   // null pointer protection
      {return;}
 var local1;  // declare a local variable
 local1 = genMat_ptr.ambient_green;
 genMat_ptr.ambient_green = genMat_ptr.diffuse_green;
 genMat_ptr.diffuse_green = local1;
 genMat_ptr.ambient_blue += 50;
}
.
.
.
```

Except for the extra if statement, it looks about the same doesn't it. We have simply changed myMat to genMat_ptr in this routine. So why would that make a difference? First lets talk about the if statement.

```
if (genMat_ptr == null)   // null pointer protection
      {return;}
```

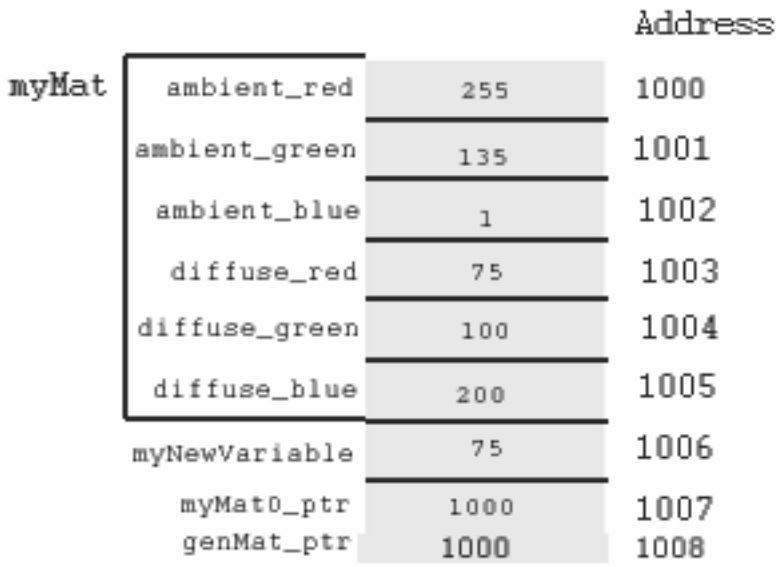
If you try to reference a null pointer as in **genMat_ptr.ambient_green =** you will get a null pointer error at run time. You don't want these occurring, so it's best to put a safety check in a procedure to make sure they don't happen. Now back to the rest of our generic routine swapMaterial().

The difference between the two routines is really how we call this function. We must load our generic pointer with the address of the material that we want to swap around.

First we can assign our generic pointer to point to the same thing myMat0_ptr points to.

```
genMat_Ptr = myMat0_ptr;  // move myMat0_ptr's value (1000) to genMat_ptr.
```

We have simply assigned the value of myMat0_ptr (1000) to genMat_ptr, a simple move. Let's see how memory looks here.



Notice that genMat_ptr has been assigned the value 1000 from myMat0_ptr. When you do an assign (=) between two pointers, that is what happens. myMat0_ptr points to myMat, so now genMat_ptr does also! Now we can call our generic swapMaterial function and let it manipulate the contents of myMat.

```
swapMaterial ();  // call our swap Material routine with genMat_ptr pointing to myMat
```

When the computer gets to a statement in our function that references a pointer
(i.e. **genMat_ptr.ambient_green = genMat_ptr.diffuse_green;**)
it knows these are pointers, and that it should dereference the address of the pointers and use the values that they point to for the move.

See the difference between the two techniques? I can now swap values around in as many material structures as I want by loading their addresses in genMat_ptr before calling swapMaterial.

Actually there is an easier way to assign genMat_ptr with the location of myMat0. We really don't even need myMat0_ptr, because the computer can directly load genMat_ptr from myMat0 like this:

```
genMat_ptr = myMat0; // assign genMat_ptr to point to myMat0.
```

This works exactly the same way, and it saves us declaring an extra pointer.

If your confused don't worry. It will sink in eventually. Try moving on to Section 7: Programming the Light Mixer (Lite) in C-Script. Follow along with the explanation. Try drawing a few pictures of what memory might look like with all the pointers. And if it still doesn't sink in, don't worry, be happy! It will come to you if you keep trying.

Section 7 - Programming the Light Mixer (Lite) with C-Script

What we will be doing here is coding a small C-Script that will allow us to adjust the lighting properties of an entity dynamically through the .material property of the entity, and through the entities own lighting effects. We will be building a 'lite' version of light mixer for learning purposes. It will include the key ideas used in Light Mixer, but some of the functionality has been left out for brevity. The full source to Light Mixer is available in the lmx.wdl file in the lightmxr folder, so feel free to use it once you get through this tutorial. It contains plenty of comments to help guide you through the code.

The main thing to keep in mind is that the Light Mixer program needs to be self contained. We don't want to have any references to anything in the 'reactor game' itself, since we want our end result to be usable for 'tuning' the lighting effects of entities in other games!

One more thing. We are going to be using a LOT of POINTERS in our code. If you don't understand the concept of pointers, be sure to read:
Section 1. The nuts and bolts of using 'Pointers' in Light Mixer

Getting Started

So let's get started. Fire up SED (the Wonderful new editor with A6 that I hope you are using!) or the editor of your choice and make a new wdl file right in the 'reactor' directory. Let's call it mylmx.wdl.

For your reference, anything in **Blue from this point on is C-script code and needs to be put in mylmx.wdl.** Anything in **Green is code we've already put in mylmx.wdl and is there only to help you line up where to insert new Blue code**. Anything in **orange is a reference to code, but it isn't necessarily even in Light Mixer.** I'm sure you already know how to type and don't need the practice, so you are welcome to simply cut and paste.

Let’s start with a header. So, cut out the Blue code below and paste it into your editor window.

```
/*****************************************************************************
Light Mixer Control Panel Script

Design goal:

Make a control panel that will let us dynamically adjust various lighting settings of
entities in a game. It should be generically written, so it can be used by any 3DGS game
developer to tweak his game scenes. Of course it does require A6 running on a system with
a 'newer' video card. It should also be as unobtrusive on the game developer’s code as
possible.
*****************************************************************************/
```

I put the design goal at the top of our program to help keep us on track. When we are done, you can be the judge of how we did.

Our First Pointer Declaration

Now we need to define a pointer that we can use to reference our target entity in the game. Because our code is going to be loaded right into the ‘games’ code via an include statement, I am going to prefix everything with lmx so we won’t have any naming conflicts.

```
/*****************************************************************************
Global Variables go here
*****************************************************************************/

entity* lmxTarget_ptr;  // pointer to the target entity
```

I like to keep all of my variables at the top. It’s an old programming habit to keep my code semi-organized. **lmxTarget_ptr** is the entity pointer that we will use to “point” to our Target Entity, (the entity that we are trying to ‘tweak’).

One of the new features of A6 is the **entity.material** functionality. This gives us a lot of control over the way an entity interacts with light. The first thing we need is to define our material structure. So add this code:

```
material lmxMat0 { }
```

I was a little surprised that this didn’t generate and error, but it turned out to be very helpful. The A6 manual defines what properties you can set up for a material, but we didn’t include any. For your reference here is what we COULD have put in the { }.

- emissive_blue, emissive_green, emissive_red
- ambient_blue, ambient_green, ambient_red
- diffuse_blue, diffuse_green, diffuse_red
- specular_blue, specular_green, specular_red
- power
- alpha
- albedo

If we define any of these Material parameters within the {}, the A6 compiler requires us to give it a value. That is well and good, but if we give it a value, this value overrides the 'default' value already set. Therefore, as soon as we attach our material structure to an entity, it changes to our settings. But I want to be able to play with the existing settings, and later on decide which ones to permanently set in my games! So, we are lucky that **Material lmxMat0 { }** works without errors.

Building our Control Panel

All of the work of manipulating the material properties is going to be handled by vertical slider panel commands. There are tutorials out there already about panels if you don't understand them. The new "3DGS_Manual.chm" manual for A6 (another great new tool!) has some good explanations of the each command we will use. Refer to the manual for command syntax and explanations.

So again, just paste away:

```
/*****************************************************************************
Panel Code for manipulating an entities lighting effect
*****************************************************************************/

bmap lmxMap = <lmxPanel.bmp>;

bmap lmxButtonRadioup = <lmxButtonRadioUp.bmp>;
bmap lmxButtonRadioDown = <lmxButtonRadioDown.bmp>;

bmap lmxButtonRightArrowUp = <lmxRightArrowUp.bmp>;
bmap lmxButtonRightArrowDown = <lmxRightArrowDown.bmp>;

bmap lmxButtonAttachUp = <lmxButtonattachUp.bmp>;
bmap lmxButtonAttachDown = <lmxButtonattachDown.bmp>;

bmap lmxSliderblue = <lmxbluesquare.bmp>;
bmap lmxSlidergreen = <lmxgreensquare.bmp>;
bmap lmxSliderred = <lmxredsquare.bmp>;
bmap lmxSliderwhite = <lmxwhitesquare.bmp>;

/*****************************************************************************
Light Mixer Control Panel
*****************************************************************************/
panel lmxPanel
{
      bmap =lmxMap;
      pos_x = 0;
      pos_y = 0; // was 175
      alpha=100;
      layer=1000;

      vslider = 8,345,120,lmxSliderred,1,255,lmxMat_ptr.emissive_red;
      vslider = 28,345,120,lmxSlidergreen,1,255,lmxMat_ptr.emissive_green;
      vslider = 48,345,120,lmxSliderblue,1,255,lmxMat_ptr.emissive_blue;

      vslider = 83,345,120,lmxSliderred,1,255,lmxMat_ptr.ambient_red;
      vslider = 103,345,120,lmxSlidergreen,1,255,lmxMat_ptr.ambient_green;
      vslider = 123,345,120,lmxSliderblue,1,255,lmxMat_ptr.ambient_blue;
```

```
      vslider = 158,345,120,lmxSliderred,1,255,lmxMat_ptr.diffuse_red;
      vslider = 178,345,120,lmxSlidergreen,1,255,lmxMat_ptr.diffuse_green;
      vslider = 198,345,120,lmxSliderblue,1,255,lmxMat_ptr.diffuse_blue;

      vslider = 233,345,120,lmxSliderred,1,255,lmxMat_ptr.specular_red;
      vslider = 253,345,120,lmxSlidergreen,1,255,lmxMat_ptr.specular_green;
      vslider = 273,345,120,lmxSliderblue,1,255,lmxMat_ptr.specular_blue;

      vslider = 308,345,120,lmxSliderwhite,1,10,lmxMat_ptr.power;
      vslider = 328,345,120,lmxSliderwhite,0,100,lmxMat_ptr.alpha;
      vslider = 348,345,120,lmxSliderwhite,1,100,lmxMat_ptr.albedo;

      vslider = 383,345,120,lmxSliderred,1,255,lmxTarget_ptr.red;
      vslider = 403,345,120,lmxSlidergreen,1,255,lmxTarget_ptr.green;
      vslider = 423,345,120,lmxSliderblue,1,255,lmxTarget_ptr.blue;

      vslider = 458,345,120,lmxSliderwhite,0,1000,lmxTarget_ptr.lightrange;
      vslider = 478,345,120,lmxSliderwhite,0,100,lmxTarget_ptr.alpha;
      vslider = 498,345,120,lmxSliderwhite,0,100,lmxTarget_ptr.albedo;

      //    vslider = 600,345,120,lmxSliderwhite,0,1000,lmxLight_ptr.lightrange;

button = 570,330,lmxButtonRightArrowdown,lmxButtonRightArrowUp,lmxButtonRightArrowUp,
           lmxGetNextEntity,null,null;

button = 600,445,lmxButtonAttachDown,lmxButtonAttachUp,lmxButtonAttachUp,
           lmxAttachMaterial,null,null;

      digits = 7,465,3,_a4font,1,lmxMat_ptr.emissive_red;
      digits = 27,465,3,_a4font,1,lmxMat_ptr.emissive_green;
      digits = 47,465,3,_a4font,1,lmxMat_ptr.emissive_blue;

      digits = 82,465,3,_a4font,1,lmxMat_ptr.ambient_red;
      digits = 102,465,3,_a4font,1,lmxMat_ptr.ambient_green;
      digits = 122,465,3,_a4font,1,lmxMat_ptr.ambient_blue;

      digits = 157,465,3,_a4font,1,lmxMat_ptr.diffuse_red;
      digits = 177,465,3,_a4font,1,lmxMat_ptr.diffuse_green;
      digits = 197,465,3,_a4font,1,lmxMat_ptr.diffuse_blue;

      digits = 232,465,3,_a4font,1,lmxMat_ptr.specular_red;
      digits = 252,465,3,_a4font,1,lmxMat_ptr.specular_green;
      digits = 272,465,3,_a4font,1,lmxMat_ptr.specular_blue;

      digits = 307,465,3,_a4font,1,lmxMat_ptr.power;
      digits = 327,465,3,_a4font,1,lmxMat_ptr.alpha;
      digits = 347,465,3,_a4font,1,lmxMat_ptr.albedo;

      digits = 382,465,3,_a4font,1,lmxTarget_ptr.red;
      digits = 402,465,3,_a4font,1,lmxTarget_ptr.green;
      digits = 422,465,3,_a4font,1,lmxTarget_ptr.blue;

      digits = 447,465,4,_a4font,1,lmxTarget_ptr.lightRange;
      digits = 477,465,3,_a4font,1,lmxTarget_ptr.alpha;
      digits = 497,465,3,_a4font,1,lmxTarget_ptr.albedo;

      digits = 600,440,2,_a4font,1,lmxMatCount;
```

```
      flags overlay,refresh;
}
```

For brevity, I didn't include all of the functionality of the Light Mixer Control Panel in this tutorial.

Here is what the final product looks like (see lmx.wdl)

If you look across the top you'll see the various color properties that we can alter with the material structure. And below each is three color coded slider knobs. With these knobs we will slide up and down the various values in the structure and OBSERVE WHAT HAPPENS. And the best part is, we can do this for any entity in the game! We'll not actually every entity, only entities that have action defined can be manipulated, but more on that later.

Our panel could directly reference any of the fields in our **lmxMat0** material structure, like this:

```
lmxMat0.ambient_red = 200; // set ambient_red property of our material (lmxMat0) to 200.
```

But I've chosen to do it a different way. We are going to reference all of the fields in our Panel with a ... Pointer! Pointers are great when you are writing generic routines, and I've got a feeling that we might want our panel to be able to 'generically' deal with multiple material structures. So, go back up to the top where we keep our global variables, and add the following (remember, only add the **BLUE** code and use the **Green** code to help line up where to put it):

```
/*****************************************************************************
Global Variables go here
*****************************************************************************/
entity* lmxTarget_ptr;   // this is our pointer to the target entity

material lmxMat0         // our material structure
{ }

material* lmxMat_ptr = lmxMat0;  // This pointer is referenced in the panel code
```

We have added another pointer, this one we will use in our 'Panel code' to reference each of the Materials properties.

So instead of directly referencing the lmxMat0 structure in our panel, we will indirectly reference the properties of our material with pointers. Like this:

```
// somewhere outside of our 'Panel' code
lmxMat_ptr = lmxMat0; // set our Panel pointer to point to our material
.
.
.
// then in our panel
lmxMat_ptr.ambient_red = 200; // effectively sets lmxMat0.ambient_red to 200
```

So why go to all this trouble? Mainly so later on we will be able to define another material, say lmxMat1, and use it with our panel without changing the panel code at all! It would look something like this:

```
Material lmxMat1 {} // Define another material

lmxMat_ptr = lmxMat1; // load up our pointer with the address of lmxMat1
.
.
.
// then in our panel
lmxMat_ptr.ambient_red = 200; // Now we are setting lmxMat1.ambient_red = 200;
```

Get it? Does this make sense? Do you see WHY a pointer is so useful?

- If no, take a break, go look over Section 6 again. Then come back here.

- If yes, congratulations, take a break, you deserve it…

Finding our 'Target' Entities

We've got our target entity pointer defined, but it doesn't 'point' to anything. We could have required the game developer to establish a pointer in his code and then pass it to us with a function but that would be too obtrusive. For our Light Mixer Control Panel, A6 has thankfully provided a better way. It's called **ent_next**. So go back to the bottom of mylmx.wdl and add this code next:

```
/*****************************************************************************
This function will cycle through all of the available entities
It sets our target pointer to the entities pointer
It also gets the string name of the entity for us to display
*****************************************************************************/

function lmxGetNextEntity ()
{
      lmxTarget_ptr = ent_next (lmxTarget_ptr); // get next entity pointer
      if (lmxTarget_ptr != null)    // null pointer protection
      {
            str_for_entname (lmxStrNm, lmxTarget_ptr);   // get the string name
            lmxStrDsply.string = lmxStrNm;  // and load it in our text display
      }
      else
      {lmxStrDsply.string = "   Entity Search";}
}
```

I'll give you a line for line breakdown of how the **lmxGetNextEntity** function works in a minute, but before I forget, we have introduced a couple of new variables to our game in this routine, so we need to go back up to the top of the global variables area and define them. So cut and paste the Blue code:

```
/*****************************************************************************
Global Variables go here
*****************************************************************************/


font arial_font = "Arial",1,12; // truetype font

string lmxStrNm [20];               // a string to hold the target entities name;
text lmxStrDsply // and this is a text panel to display the name of the target entity
{
      layer = 1000; pos_x = 534;pos_y = 315;
      font = arial_font; string = "   Entity Search";
      alpha = 100;flags = center_x, narrow, transparent;
}
```

Here we have added a string to store the entities name in, and a text panel to display it with. We've also added a font type of arial_font.

How lmxGetNextEntity () Works Line by line

ent_next gives us access to an A6 internal list of every entity in our game. And, better yet **it returns a pointer to an entity**! To use it, you pass it an entity pointer and in return it gives back to us the next entity on the list. But wait, we said we don't have our entity pointer loaded, therefore it is NULL. Well, if you pass **ent_next** a NULL pointer, it returns to you the first entity in it's internal list! Now for the line for line breakdown.

```
function lmxGetNextEntity ()
{
      lmxTarget_ptr = ent_next (lmxTarget_ptr); // get next entity pointer
```

Since **lmxTarget_ptr** is null the first time we call this procedure, we will get back the very first entry in the table and store it in **lmxTarget_ptr**! Ok, so what about this:

```
      if (lmxTarget_ptr != null)    // null pointer protection
```

Why do we need null pointer protection if we know it's going to return us the next entities pointer? The answer is in the manual. If you get to the last entry in the table, **ent_next** will return a NULL pointer. So this one line of code when repeatedly called will cycle us through the entire list of entities until it reaches the end, and then start all over again! Pretty nice functionality for one line of code!

```
      {
            str_for_entname (lmxStrNm, lmxTarget_ptr);   // get the string name
```

str_for_entname returns the Entity Name used in WED. The default name is the model name you are using with a number attached, but you can change it to whatever you like in WED by using the properties box. I tried using **lmxStrDsply.string** as the target string for **str_for_entname** but it the A6 compiler wouln't allow it, therefore I had to make another string called **lmxStrNm**.

```
            lmxStrDsply.string = lmxStrNm;  // and load it in our text display
      }
```

And of course the above line simply puts the string name into the display panel's string for us.

Attaching to the entity

Now that we have our entity pointer all loaded up, along with the entity name, we need some code to attach our material structure to it. We'll use the same technique as the last time, I'll give you the entire function all at once, then we'll go add any new variables and finally we'll go through it line for line. So go to the **BOTTOM of mylmx.wdl** and paste away!

```
/*************************************************************************
Function to attach materials to our target
*************************************************************************/
function lmxAttachMaterial ()
{
      if (lmxMatCount > 1)    // test to see if we are already attached
            {beep;return;}    // and sound off if we are

      if (lmxTarget_ptr == 0)  // null pointer check
            {beep;return;}

      lmxTarget_ptr.material = lmxMat_ptr; // attach our Generic Material Pointer

      if (lmxMatCount > 0) {lmxMatCount -= 1;}         // count down
}
```

We seem to have added a variable in this procedure, so lets go back up to our globals section and add it in. Paste only the **Blue** code:

```
/*************************************************************************************
Global Variables go here
*************************************************************************************/
entity* lmxTarget_ptr;               // this is our pointer to the target entity
font arial_font = "Arial",1,12; // truetype font
string lmxStrNm [20];               // a string to hold the target entities name;
text lmxStrDsply // and this is a text panel to display the name of the target entity
{
      layer = 1000; pos_x = 534;pos_y = 315;
      font = arial_font; string = "   Entity Search";
      alpha = 100;flags = center_x, narrow, transparent;
}

var lmxMatCount = 1;
```

How lmxAttachMaterial () works

```
/*****************************************************************************
Function to attach materials to our target
*****************************************************************************/
function lmxAttachMaterial ()
{
      if (lmxMatCount > 1)    // test to see if we are already attached
            {beep;return;}    // and sound off if we are
```

Here we are simply testing our Material Counter to see how many Materials we have left to attach to entities. Actually with our 'lite' version of Light Mixer we will only be able to attach material to one entity. However if you want to see how I did it for multiple entities, you can refer to the complete versions source in lmx.wdl.

```
      if (lmxTarget_ptr == null)  // null pointer check
            {beep;return;}
```

And here we are checking to make sure we don't have a null pointer as our target entity. It's always a good idea to check our pointers before using them.

```
      lmxTarget_ptr.material = lmxMat_ptr; // attach our Generic Material Pointer
```

And of course, this is the meat of the function. We are setting what our target entity's material (pointed to by **lmxtarget_ptr**) to our material **lmxMat0** (pointed to by **lmxMat_ptr**).

```
      if (lmxMatCount > 0) {lmxMatCount -= 1;}         // count down
}
```

And finally we decrement our count of the maximum number of entities that we can attach material to.

Displaying our Panel

There's not much left to do. We need to make our panel visible. Here is the function to do that. Go to the bottom of mylmx.wdl and add the following:

```
/*************************************************************************
Raise and lower the Light Mixer Control Panel
*************************************************************************/
function lmxTogglePanel
{
      lmxPanel.visible = (lmxPanel.visible == Off); // toggle our panels visibility
      LmxStrDsply.visible = (lmxPanel.visible != Off); // match lmxStrDsply visibility
}
```

What's this? If you know C, then you recognize its elegant style. If not, this looks kind of weird. We are doing a == (compare for equal), but where is the IF statement? Here is how this works.

How lmxTogglePanel works

First let's look at the stuff in the ().

```
(lmxPanel.visible == Off);
```

This is indeed a compare. We are comparing the value of lmxPanel.visible to the defined value of off. Well in most computer languages FALSE is represented by the value zero and TRUE is represented by any non-zero value. C-Script OFF and ON are defined in the same as TRUE and FALSE. That is Off is zero and On is 1.

When you put something in () you are saying evaluate what's in the () and return a Boolean (TRUE (1) or FALSE (0). So the computer evaluates the visible property of **lmxPanel** and returns 1 or 0. Lets take the two cases of what visible can be.

If Visible is	(visible == OFF) returns
0	TRUE (1)
1	FALSE (0)

Notice the toggling effect! If 0 then 1. If 1 Then 0.

`lmxPanel.visible = (lmxPanel.visible == Off)` **is the same as saying**

```
If (lmxPanel.visible == Off) {lmxPanel.visible = On;} else {lmxPanel.visible = Off;}
```

Of course this next line is basically the same thing, except rather than toggling we want the visibility of ths lmxStrDisplay to be the same as the visibility of the lmxPanel.

```
LmxStrDsply.visible = (lmxPanel.visible != Off); // match lmxStrDsply visibility
```

Try using this C style in other ways. You'll like it once you get use to it. If you really like this concept, they have contests each year for C programmers to see who can do the most work in the least # of CHARACTERS. Not lines of code, but total CHARACTERS. You would be amazed at what they come up with.

Finishing Up

Well we only have one line left for mylmx.wdl. We need to be able to press a key and run the **lmxTogglePanel ()** function. I picked the letter p (for panel). So here is the last line of code in mylmx.wdl. Paste it at the bottom.

```
on_p = lmxTogglePanel;
```

Finally, if you are using the 3DGS editor, SED, you might want to find the “Indent All” symbol in the middle of the toolbar and press it to fix up the indentation.

That’s it, save and close your mylmx.wdl file.

Using mylmx.wdl with Reactor

Now we need to make a small change to the room.wdl file in the reactor folder to use your code. Find the line that says:

```
include <lmx.wdl>;            // Light Effects Panel
```

and replace it with our new code:

```
include <mylmx.wdl>;
```

That’s it. Give it a try. Run the reactor game. Press p to activate the panel. Right click once to activate the mouse pointer. Click on yellow arrow in the upper right hand corner of the panel until you see ‘reactor1’. Click on the blue attach button to attach our material to the reactor, and then tweak away.

Remember this is the ‘lite’ version of Light Mixer, so it only allows you to attach 1 material to 1 entity on any run of the game. It also does not allow you to toggle On and Off the various entity flags like transparent, bright, metal.

If you want to see how to do those things, and more, take a look at lmx.wdl in the lghtmixer folder. There are plenty of comments to help you out.

I hope you like it. Have fun with 3DGS A6 and LightMixer.

Send comments to Giorgi3@mchsi.com