Acknowledgement

Thanks a lot to ‘res’ for answering the many questions that enabled me to terminate this tutorial.

Shader 101

The goal of this tutorial is to show how shaders works through a little example requiring some shaders manipulation. We will built a silly application that hilite meshes when clicked. It is assumed that the reader has some knowledge of ‘Cg’ and Cs in general. For Cs, make sure you went through the simmap tutorial. From that tutorial, you should get started on:

- How to find mesh under mouse clicks
- Accessing mesh shaders parameters
- Changing mesh shaders parameters
- Shader writing using Cg
- Some information on ‘how to debug that darn shader’
- Very basic usage of shaders in blender and blender2crystal

Bills of material

This tutorial has been written using crystal space 1.0, Blender 2.43 and ‘blender2crystal’ 0.6.0. It was tested on a windows XP laptop with an ATI Mobility X700. Any other OS/Machine combination should work, but this has not been verified. If there are specifics to machines/OS, do not hesitate to update this tutorial. To download those items, go to:
Crystal space: http://www.crystalspace3d.org/main/Download
Blender: http://www.blender.org/download/get-blender
Blender2crystal : http://b2cs.delcorp.org/index.php/Main_Page

Principle of hiliting

Typically, hiliting is done either by multiplying the final color of a pixel by some factor or by adding some constant value which has almost the same effect.
So, if we are speaking ‘Cg’in the shader, we will have something like that in the fragment program code

```
float4 FragmentMain (…,uniform float4 hiliteColor, …) COLOR {
…
    color.rgb += hiliteColor;
…
    return color;
}
```

In this code, hiliteColor is a variable coming from the application. Because it is uniform it is entirely under the control of the application, the shader cannot modify it in any way. Assuming we have a shader that implements that kind of hiliting, we simply need a way in the application to access the hiliteColor variable used inside the fragment. This is done using a ‘shader variable’ that will be stored in the material of the mesh.

Background information

The crystal space render structure is based around the ‘renderloop’. This is the mechanism that sequence the rendering of things, like ‘lighting’, ‘shadows’, specify a default shader. Renderloops are behond the scope of that tutorial, but lie in the background.

Accessing material shader variables

Shader variable are accessible on objects that implement the ‘iShaderVariableContext’ interface. For material, this is accessible via the ‘iMaterial’ interface. The material of a mesh can be accessed through the following relations, starting from the engine:

```
iEngine->iMeshWrapper->iMeshObject->iMaterialWrapper->iMaterial
```

The iShaderVariableContext interface provides access to the ‘csShaderVariable’ objects. The shader variable system is fairly versatile and can hold a wide range of data types. Check the api reference for ‘csShaderVariable’. We will use the name ‘mat hilitecolor’ for this variable (the space is allowed). Then to modify its value on a material, we can do the following:

```
    iStringSet r_CSStrings=CS_QUERY_REGISTRY_TAG_INTERFACE
(object_reg,"crystalspace.shared.stringset", iStringSet);
    StringID id_shader_mat_hilite_color = r_CSStrings->Request("mat
hilitecolor");
    csShaderVariable *pHiliteColor = p_iMaterial-
>GetVariableAdd(id_shader_mat_hilite_color); // … The shader variable
    pHiliteColor->SetValue(csColor(0.3f,0.3f,0.3f));
```

Since we assumed the shader understood that variable, we should get an hilite effect.
For a reference of other shader variable provides look in documentation at the following place: http://crystalspace3d.org/docs/online/manual/Shader-System-Overview.php#0. For our usage, we will set the variable on meshes.

Lab 1: Modifying simmap to hide meshes on mouse click

First we will create an application that hides mesh when we click on them. If this bore you to death, the code is already there and you can skip to the part where we will hilite things.

The solution for this part is in is in the directory ‘Solution for lab 1’.

- What we do here is create an external application.
- Implement code that will hide stuff when clicked.
- If this is not of interest, you can start from here and go to Lab 2 directly

Step 1: create an external application duplicating the ‘simpmap’ code.
Compile and run, it should find the same simpmap start-up world inside crystal space automatically thancks to VFS magic. Check ‘creating external application’ tutorial for that purpose. Microsoft visual studio little known trick (at least to me …) : you can create pre sets properties in the view called ‘property manager’. Then in your project settings of an empty project, in the ‘general section’ ‘Inherited project property sheets’ , set a path to that property vsprops file. Nothing else to set (but the inherited settings don’t show up in the properties). I created such a preset in the file named ‘Crystal-debug.vsprops’ usable for any projects.

Step 2: Modify the ‘LoadMap’ method to load the simple world provided for this tutorial (3 spheres with some lights)

```
bool Simple::LoadMap ()
{
  // Set VFS current directory to the level we want to load.
  csRef<iVFS> VFS (CS_QUERY_REGISTRY (GetObjectRegistry (), iVFS));
  // VFS->ChDir ("/lev/castle");
```

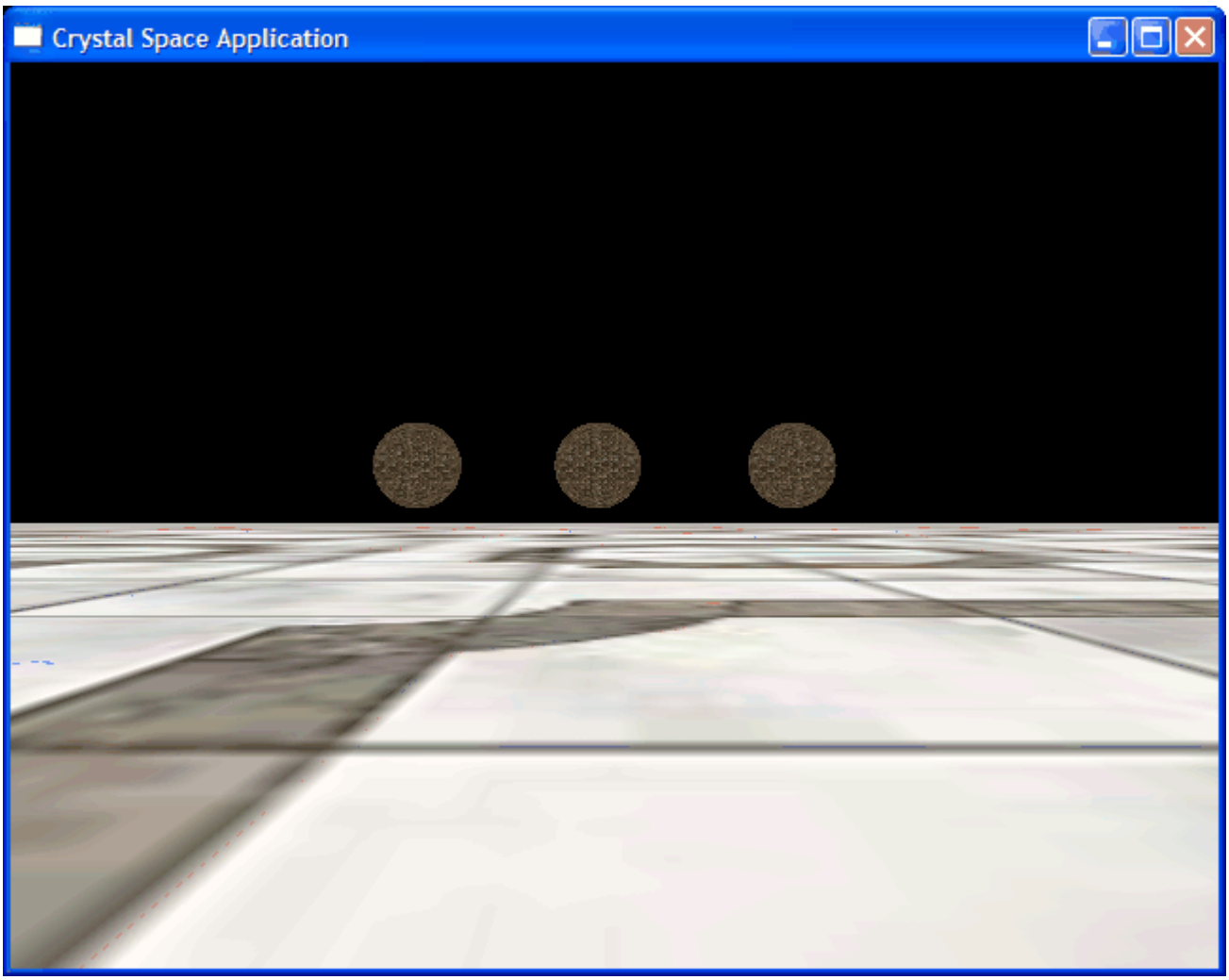
```
  // Use ChDirAuto to use 'real' path.  In that case a window path local
  // to the launch dir of the application.
  VFS->ChDirAuto(".\\world.zip");
  // Load the level file which is called 'world'.
  if (!loader->LoadMapFile ("world"))
    ReportError("Error couldn't load level!");

  return true;
}
```

We use the VFS->ChDirAuto method instead of ChDir to load file using native platform paths. OF course, it is possible use the VFS independent notation by mounting the world, but the functionality to load native file paths is not so much advertised. So, here is how one can load native file paths using VFS to do all the work automatically (specifically, it automatically detect it is a zip file).

Step 3: Copy the world file in the appropriate location

copy the ‘world.zip’ file provided with this tutorial in the directory where the application will run (available in blender and CS xml format in ‘Solution for lab 1’, data/world.zip and data/ShaderTutorial.blend). If using Microsoft Visual C++, make sure to set the ‘Working Directory’ debugger preference to that directory.
Exercise for the reader that never really used blender2crystal: export the world from blender rather than using the one provided.
Compile and run, it should produce this (can move around slowly with arrows):



Step 4: Add code to Hide/UnHide by overriding 'OnMouseDown' and 'OnMouseUp' and adding a dummy 'HiliteMesh' and 'UnHiliteMesh' method that simply set/reset the invisible mesh flag:

```
void Simple::HiliteMesh(iMeshWrapper *mesh) {
    // Hide the mesh.
    mesh->SetFlagsRecursive(CS_ENTITY_INVISIBLE);
    hilitedMesh = mesh;
}

void Simple::UnHiliteMesh() {
    if ( ! hilitedMesh ) {
        return;
    }
    // Show the mesh.
    hilitedMesh->SetFlagsRecursive(CS_ENTITY_INVISIBLE,0);
    hilitedMesh.Invalidate() ;
}

//
// Perform our mouse down actions.
// In this case, it concists only in finding who was under the mouse click,
// and call the routine to hilite our mesh.
//
```

```
bool Simple::OnMouseDown (iEvent &event) {
    csVector3       ;
    int32          mouse_x = csMouseEventHelper::GetX(&event);
    int32          mouse_y = csMouseEventHelper::GetY(&event);

    csScreenTargetResult result = csEngineTools::FindScreenTarget(csVector2
(mouse_x, mouse_y),1000.0f,view->GetCamera());
    if (result.mesh) {
        HiliteMesh(result.mesh);
    }
    return false;
}

bool Simple::OnMouseUp(iEvent &event) {
    UnHiliteMesh();
    return false;
}
```

Compile and run, clicking on anything will hide/unhide it.

Lab 2 : All the required item to hilite, without the shader

Step 1: add a class called 'csMeshHiliteHelper' that will perform the hiliting. The code is bellow:

It simply implement access the shader variable context on the mesh to set the shader variable.

csMeshHiliteHelper.h:

```
#ifndef __CSMESHHILITEHELPER_H__
#define __CSMESHHILITEHELPER_H__

#include <crystalspace.h>

class csMeshHiliteHelper {
private:
    static csStringID    id_shader_mat_hilite_color ;

public:
    static void setMeshHiliteColor (iObjectRegistry* object_reg, iMeshWrapper
*mesh, const csColor &color );
};

#endif
```

csMeshHiliteHelper.cpp:

```
#include "csMeshHiliteHelper.h"

csStringID csMeshHiliteHelper::id_shader_mat_hilite_color = csInvalidStringID;

void csMeshHiliteHelper::setMeshHiliteColor(iObjectRegistry* object_reg,
iMeshWrapper *mesh,const csColor &color) {
    if ( id_shader_mat_hilite_color == csInvalidStringID ) {
        //
        // Fetch the string id for the shader variable we will use
        //
        csRef<iStringSet> csStrings = CS_QUERY_REGISTRY_TAG_INTERFACE
(object_reg,"crystalspace.shared.stringset", iStringSet);
        id_shader_mat_hilite_color = csStrings->Request("mat hilitecolor");
```

```
    }
    csShaderVariable *pHiliteColor = mesh->GetSVContext()-
>GetVariableAdd(id_shader_mat_hilite_color);
    pHiliteColor->SetValue(color);
   }
```

Be aware that the string set from which we search our string id must be the same used by the shader plugin when loading shaders, otherwise the Id will not be consistent. Specifically the cel stringset provided by default to behaviour is an other one.

An important note: this code is too simple. This will work on genmesh. For thingmesh, it will crash (all the meshes in the example are genmesh). As information, the example code also include the a call for setting material shader variables.

Step 2: Call the hilite routines from HiliteMesh/UnHiliteMesh:

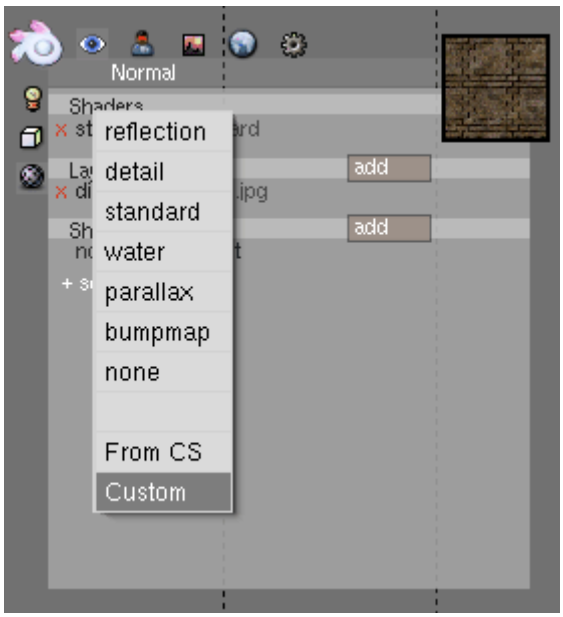
```
void Simple::HiliteMesh(iMeshWrapper *mesh) {
    UnHiliteMesh();

    csMeshHiliteHelper::setMeshHiliteColor(GetObjectRegistry(),mesh,
csColor(0.3f,0.3f,0.3f));
    hilitedMesh = mesh;
}

void Simple::UnHiliteMesh() {
    if ( ! hilitedMesh ) {
        return;
    }
    csMeshHiliteHelper::setMeshHiliteColor(GetObjectRegistry(),hilitedMesh,
csColor(0.0f,0.0f,0.0f));
    hilitedMesh.Invalidate() ;
}
```

Step 3: Modify the map file to point to the shader we will write

- Either edit the ‘world’ file directly (quite error prone) or
- load the blender file from Lab 1 ‘ShaderTutorial.blend’ in blender
- Select each meshes and choose to specify a ‘Custom’ shader in the ‘Shader’ section and set it to ‘/this/diffuse_cg.xml’:

- 

- Export the file into CS. An example result is provided along with the code in ‘Lab 2 solution/data/world.zip’along with the blender file.

Step 6:compile and run.

You will see CS fetch the default shader defined by the default render loop and complain that our shader is missing.

Lab 3: Implementation of an extremely simple shader with hilite capabilities

The shader system has been designed to be quite versatile. The general structure of a shader is based on a collection of technique, each with a priority. A technique is then a collection of pass, each pass including some code to do the rendering for that pass. That rendering is delegated to plugins. The plugins available are Cg, ps1, arb, fixed gl. The largest priority technique that is usable on a given hardware is used. We will use Cg.
Then, an XML template mechanism is provided. It enables one to alter the exact XML code obtained depending on some configuration data, hardware capabilities or presence/absence of some variables in the object being rendered.

Step 1 : A minimal shader with Cg

The code is the following for a simple shader that output the texture color:
File ‘my_shader/diffuse_cg.xml’ in Solution for lab 3:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- The hello world of shaders: just return the diffuse texture. -->
<shader compiler="xmlshader" name="diffuse_cg">
<technique priority="200">
  <pass>
      <buffer source="texture coordinate 0" destination="texture coordinate
0" />
      <vp plugin="glcg" file="/this/my_shader/diffuse_cg.cgvp" />
      <texture name="tex diffuse" destination="TexDiffuse" />
      <fp plugin="glcg" file="/this/my_shader/diffuse_cg.cgfp" />
  </pass>
</technique>
</shader>
```

Comment:
We declare a shader named ‘diffuse_cg’. This is the name referenced in the world file. Blender2crystal stripped the directory path and the file name extension to compute the shader name. We create a binding with the ‘buffer’ statement that will map the mesh ‘texture coordinate 0’ buffer to the ‘texture coordinate 0’ destination. The ‘texture coordinate X’ destinations correspond to TEXCOORDX in the Cg code. X=0 correspond to the classic texture coordinates. It is not needed to define the position buffer. It is there by default (even though we don’t use it in the fragment, it is mandatory to compute the transformed position). We then define a texture sampler with the ‘texture’ statement and map that to the destination ‘TexDiffuse’. The ‘tex diffuse’ corresponds to the ‘texture’ statement in the XML world format and is available on all models. The ‘TexDiffuse’ destination corresponds to the variable we use in the Cg fragment program. Then, the Cg code is rather straight forward:

The file ‘diffuse_cg.cgvp’ containing the vertex shader:

```
<cgvp>
<program>
<![CDATA[

struct app2vertex {
   float4 Position : POSITION;
   float2 TexCoord : TEXCOORD0;
};

struct vertex2fragment {
   float4 Hposition : POSITION;
   float2 TexCoord : TEXCOORD0;
};

vertex2fragment main (app2vertex IN,
   uniform float4x4 ModelViewI : state.matrix.modelview.inverse,
     /* The matrices are bound with the state.matrix.... semantics.
        For more info see the Cg and ARB_vertex_program docs.
        */
   uniform float4x4 ModelViewProj : state.matrix.mvp)
{
   vertex2fragment OUT;

   OUT.Hposition = mul (ModelViewProj, IN.Position);
   OUT.TexCoord = IN.TexCoord;

   return OUT;
}

]]>
</program>
</cgvp>
```

The file ‘diffuse_cg.cgfp’ containing the fragment shader:

```
<cgfp>
<program>
<![CDATA[
struct vertex2fragment {
   float4 Hposition : POSITION;
   float2 TexCoord : TEXCOORD0;
};
```

```
float4 main (vertex2fragment IN,
        uniform sampler2D TexDiffuse) : COLOR
{
   return tex2D (TexDiffuse, IN.TexCoord);
}

]]>
</program>
</cgfp>
```

It is not mandatory to cut the vertex shader and the fragment shader into separate files. We could have included the XML here in the 'diffuse_cg.xml'. I find it more convenient.

Step 2 : compile and run … in case of black screen …

Since we write shader, a syntax mistake or any mistake will likely result in a black screen. To help debug this, one can do the following:

add option '--cfgfile=/config/shader-debug.cfg' which contains switches for the Cg plugin to dump the assembly generated (should work on any hardware). If you see no mention of the vertex shader being dumped, then the vertex shader failed to compile. Same for fragment shader. The available switches are described in the file /config/shadermgr.cfg in the CS distribution.

- If you can't figure out anything, do not provide any vertex shader and verify something happens in the fragment shader at all (for example, simply return a constant color at the end of the code to check that the fragment compiles).
- Use the many shader in the crystal space data/shader as examples to help debugging.
- Eventually, run the Cg code inside the Cg compiler by hand to verify what is happening.

Step 3: Add the 'hilite' functionality:

We add the shader variable. This is done using the 'variablemap' statement that must be within the cgfp element of the XML where the uniform variable will be used.
So, only the diffuse_cg.cgfp file changes:

```
<cgfp>
<variablemap variable="mat hilitecolor" destination="hiliteColor" />
<program>
<![CDATA[
struct vertex2fragment {
   float4 Hposition : POSITION;
   float2 TexCoord : TEXCOORD0;
};

float4 main (vertex2fragment IN,
        uniform sampler2D TexDiffuse,
         uniform float4    hiliteColor) : COLOR
{
   return hiliteColor + tex2D (TexDiffuse, IN.TexCoord);
}

]]>
</program>
</cgfp>
```

Re-run the program. If everything went right, you will see a weird effect: when clicked for the first time, everything is hilited, not just the mesh under the mouse. The reason is rather obvious once you thinck into it: Open GL is state based. Once a state is set, it stays there between

primitive rendering. The ‘hilite color’ behaves just like this. When clicked on the first instance, the state is set, but then stays there. Clicking on other meshes will add the state variable theres too, and ultimately it will work has expected.

Step 4: use template mechanism to generate a different shader depending on variable presence

We use the template mechanism which provides a solution to check the presence of a variable, and do something or not based on that knowledge. It is to be noted that the condition is verified dynamically in a very efficient way. The shaders needed are generated/recompiled only once thancks to a caching mechanism.
Using that method, the fragment shader diffuse_cg.cgfp become:

```
<cgfp>
    <variablemap variable="mat hilitecolor" destination="hiliteColor" />
    <program>
        <![CDATA[
struct vertex2fragment {
float4 Hposition : POSITION;
float2 TexCoord : TEXCOORD0;
};

float4 main (vertex2fragment IN,
uniform sampler2D TexDiffuse,
uniform float4    hiliteColor) : COLOR
{
        ]]>
<?if vars."mat hilitecolor"?>
        <![CDATA[
return tex2D (TexDiffuse, IN.TexCoord) + hiliteColor;
        ]]>
<?else?>
        <![CDATA[
return tex2D (TexDiffuse, IN.TexCoord);
        ]]>
<?endif?>
        <![CDATA[
}
        ]]>
    </program>
</cgfp>
```

Comment:

- We must close/open the CDATA escape section to allow the parser to see the template expression <?if …?>
- If you specified the –cfg option to get some debugging info, you will see that a second shader is compiled. The shader plugin place in the folder <temp>/shader the result of the template expansion. On windows, the temp folder is located at C:\Documents and Settings\<your user name>\Local Settings\Temp.

Step 6: I leave as an exercise to me and to the reader to:

write the same shader using the ‘fixedfunction’ shader and add the necessary technique selection.

modify the Crystal space ‘std_lighting.xml’ (and the many includes) to do the same thing than here.