USES: Uniview's Shader Effect System

Mir Jawad Mairaj

2013-02-25
USES: Uniview's Shader Effect System

Examensarbete utfört i Medieteknik
vid Tekniska högskolan vid
Linköpings universitet

Mir Jawad Mairaj

Handledare Camilla Forsell
Examinator Matt Cooper

Norrköping 2013-02-25
Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Mir Jawad Mairaj
Uniview Shader Effect System (USES)

Author: Mir Jawad Mairaj

Supervisor: Staffan Klashed

Examiner: Matt Cooper
List of Figures

Figure 1: Block diagram showing structure of a USES Module.........................9
Figure 2: Structure of an Effect Node Script................................................. 10
Figure 3: A typical mesh configuration script describing a car....................11
Figure 4: Effect node configuration of a rainbow effect..............................31
Figure 5: A Rainbow Effect on Earth..........................................................32
Figure 6: Shadow Mapped Hubble Model................................................. 32
Figure 7: Earth Glow on Hubble.................................................................32
Figure 8: Shadow Mapped ISS Model with Earth Glow..............................33
Figure 9: Earthquake Data Visualization 1..................................................33
Figure 10: Earthquake Data Visualization 2............................................. 34
Figure 11: Earthquake Data Visualization 3..............................................34
Figure 12: Effect Node configuration of an earthquake data visualization.....35
Figure 13: An illustration of the orbital elements for the asteroid data set (image courtesy of wikipedia)..............................................................35
Figure 14: Asteroids Visualization 1............................................................36
Figure 15: Asteroid Visualization 2..............................................................36
Figure 16: Effect Node configuration of an asteroid data visualization.........37
Figure 17: Funky effect created by changing aspect ratio in Uniview...........37
Index of Tables

Table 1: Uniview's Statemanager property types that can be changed through effect nodes
........................................................................................................................................ 18

Table 2: Shader constants that are automatically defined for a shader based on the material of a sub mesh
........................................................................................................................................ 21

Table 3: Configurable GL States in USES...............................................................................24

Table 4: Available textures in the shader based on the material of the sub mesh
........................................................................................................................................ 26

Table 5: Available texture modifiers in the shader based on the material of the sub mesh
........................................................................................................................................ 27

Table 6: Available material parameters in the shader based on the material of the sub mesh
........................................................................................................................................ 27
I would like to thank the staff at SCISS for providing an amazing productive and friendly environment to do my thesis work. In particular Staffan Klashed (my external supervisor) for making sure my work is on track and Marcus Lysen for providing timely inspiration and guidance.

I would also like to thank Carter Emmart (AMNH) for his role in setting up the thesis work with support from SCISS and Linköping University, my internal supervisor Matt Cooper for reviewing the work and providing key feedback for my thesis.

Lastly I would like to thank my parents and my brother for being a source of comfort and resilience.
Abstract

This thesis work details the implementation of a generic shader effect system for Uniview (a proprietary 3D graphics software for the display of astronomical data in digital planetariums developed by SCISS AB). The system enables Uniview to load a variety of 3D file formats, customize them in various ways and, using a multipass setup, create complicated effects including rainbows, shadows, orbiting meteorites etc. The functionality is similar (though tailored to Uniview's need) to effect systems such as AMD's RenderMonkey or Nvidia's FX Composer.
1 Introduction

Uniview [6] is a 3D visualization software system for digital planetariums for showcasing astronomical datasets. It has been developed by SCISS AB. As with any small development company the need to deliver new features quickly versus quality is always a battle. In order to get different scientific data sets for visualization Uniview has, hitherto, used various interfaces. Though the outcome could be effective, the customization was limited through pre-defined parameters. A custom change by a customer would take some tweaking and development of a new data set visualization feature would require a long development cycle. Furthermore, there was a growing need from the planetarium community for the ability to load various 3D digital assets (3D models) of varying formats. Uniview at the time only supported limited 3D meshes of .fbx format with little support for customization.

Another feature that Uniview lacked hitherto was a generic shader effect system that could be used quickly for prototyping a new data set and iterating over it to make it into production level quality. Also while SCISS has traditionally provided the best possible customization support to its users, the interface to the user had to be a specific solution to a particular problem and no generic framework existed that could incorporate a larger subset of the available options.

That is where USES (Uniview's Shader Effect System) comes in. Through USES users can import a variety of 3D mesh formats into the product, customize the materials, shaders etc. In addition, by combining various meshes, interesting effects like sunglare, rainbows, light shafts etc. can be created quickly through USES scripts.

1.1 Overview

The following section outlines the thesis. I will begin by summarizing the feature set for USES. Chapter 2 describes the "Effect Nodes", which are the building blocks for USES. Chapter 3 describes the interface of the USES script file which is used by the user to define effects. Chapter 4 shows some visualization results (courtesy of SCISS AB) using the system. Chapter 9 draws conclusions from the work and details directions for future development of USES.

1.2 USES Feature Set

Uniview's Shader Effect System, or USES for short, is a shader effect system similar in vein to AMD's RenderMonkey [1] or Nvidia's FX Composer [2]. It supports GLSL as the shading language and supports various 3D formats such as collada, .3ds, .x, .obj, .fbx etc. The idea of a generic shading effect system is that it raises the abstraction level for creating visual effects from a software development standpoint. It acts like a software engine, the user of which need not care about the underlying implementation and so speeds up the development cycle. A technical developer could implement an effect and let the artist tweak the parameters until satisfied with the visual outcome. Hence one effect (like a generic particle framework) can be re-utilized by artists for various purposes.

Following are some of the features supported by the system:
• Supports many open 3D formats like .3ds, .obj, .stl, .dae etc based on the opensource library asset import [3] and .fbx format using the autodesk fbx sdk.
• Ability to customize mesh material either by changing its default material properties/textures or specifying a custom GLSL shader.
• Ability to control GL states for each rendering of a sub-mesh. For example changing the blend/scissor/stencil etc. states for each material of the mesh differently.
• Loading .dds cubemaps giving flexibility for some effects and loading of .dds 3D textures. An immediate benefit is to get noise in the shaders.
• Development hints for faster debugging like: warning on type mismatch for uniform variables in shader with custom specified variable in scripts, warning on unused materials specified in configuration files, ability to view the multiple passes output in different viewports etc.
• Support for animation with or without skinning based on supported formats by open asset import library [3].
• Multiple passes supported with ability to render multiple objects in each pass.
• Support for rendering to texture.
• Ability to load and switch between different effect nodes at runtime.
2 USES Implementation

Uniview have various module types for different datasets like planets, stars, galaxies, constellations, satellites, etc. All modules in Uniview follow a similar approach: A module file (which gives the scene specification), a configuration script and data files. USES follows a similar design pattern with an added advantage of having the ability to have different configuration scripts known as Effect Nodes configuration scripts.

USES extended organically at first from a simpler mesh module in Uniview. The earlier mesh module could load a single .fbx 3D model and a user could specify a vertex and fragment shader written in CG shader language. In USES however we improve and add newer capabilities as will be described next.

2.1 Effect Nodes

A Uniview USES module can have multiple effect nodes attached to it and the user can switch between them at runtime. In this section I will describe an overview of various aspects of an effect node.

![Figure 1: Block diagram showing structure of a USES Module](image-url)
Figure 2 depicts the structure of an Effect Node in USES which details a USES configuration script. The red circles depict components which are essential to a configuration script. In Chapter 3 the keywords will follow the same color coding structure to distinguish the different stages.

In the configuration file we specify which 3D model(s) to load. We can also make changes to the model material that will override the default behavior. The 3D model is loaded with the asset import library[3]. It can have a set number of textures and have different material properties per sub-mesh. Using the mesh configuration we can modify each and every material present in the mesh.

To see what materials are present in a mesh we can use the open asset import viewer utility[4].

By “modification to default behavior” we mean:
1. Specifying a custom shader to be used for a particular material or all materials.
2. Override the default textures of a material from the model or disable them.
3. Override the material properties of the default material properties present in the 3D model.
4. if we use custom shaders then we can specify our own uniform variables (vectors, arrays, matrices, texture samplers etc)
The basic filtering of the kind of rendering on sub-meshes of an object is based on the material type. We change the material behavior to effect the change in rendering of a sub-mesh. This can be clarified with an example of a typical mesh configuration script show in Figure 3.

```plaintext
mesh
{
  radius 1000
  data car ./modules/ship/car/car.3ds
  enableTypeCheck true
  glslVersion 1.5

  pass
  {
    useDataObject car
    passEnable true
    passScale 1
    shader
    {
      type defaultMeshShader
      {
        Mtrl defaultMeshMaterial
        TexFlipV false
        TexFlipU false
        define SG_DIFFUSE_TEXTURE
        # use the diffuse texture from the loaded model (if it has any)
        define SG_REFLECTION_TEXTURE
        texture reflectionTexture ./modules/Ship/Reflect.bmp
        parameterIf reflectionTexStrength 1
        parameterIf1 reflectionTexOp 1
        define SG_NORMAL_TEXTURE
        texture normalTexture ./modules/Ship/marine_local.tga
        parameterIf opacityMtrl 0.25
        parameterIf shininessMtrl 50
      }
      type defaultMeshShader
      {
        Mtrl chassis
        vertexShader ./data/shaders/meshsimple.vs
        fragmentShader ./data/shaders/meshsimple.fs
        parameter3f ambientMtrl 0.0 0.0 0.0
        parameterIf opacityMtrl 1
        SceneTransformVar c_Aqua Earth earthpos 0 0 0
        StateManagerVar orbitcamera.Distance emissiveMtrl
        GL_State
        {
          UV_BLEND_ENABLE true
          UV_BLEND_COLOR 0.1 1 0.1 0.3
        }
      }
    }
  }
}
```

Figure 3: A typical mesh configuration script describing a car

The object loaded is that of a car. It has two materials specified. So those sub-meshes that use “chassis” material will use the second shader configuration and for all other material the first shader configuration (mtrl = defaultMeshMaterial) will be used.
As seen from the Figure 2 the effect node consists of two stages.

Stage 1 is the setup stage: for data to be used, for rendering textures of render targets, for customizing animation sequences, for setting up of state manager properties and other parameters. Stage 2 is where the actual drawing sequence of the effect is defined. The drawing is divided into different passes.

The following subsections take a deeper look in each component of the effect node and illustrate the purpose of different commands that are used in it.

2.2 Stage 1

2.2.1 Various Configuration Parameters for Stage 1

The first thing is the cull radius of the effect. This radius is specified in the units of the USES module. Also every object used in the data mesh is scaled by default so that its bounding radius is that of the cull radius specified.

Another option involves specifying which version of GLSL should be used to compile the shaders used in the effect node. This is useful for compiling effects for different computers with varying levels of GLSL support.

Another option is a debug warning option, which is turned on by default, that involves checking the types of variables provided in the effect node (USES) configuration script with their counterpart used in the GLSL shaders. It is recommended to keep this option on.

2.2.2 3D Data File List

The user has to also specify the data items (3D mesh files) that will be referenced in Stage 2.

2.2.3 Animation Configuration for 3D Data

Given the 3D data, some can include animations. By default the animation played for the objects would play the first sequence from start to finish using the default time step described in the mesh data file, but this can be customized at this point for each 3D object.

2.2.4 Renderable Texture List

Here one specifies all renderable textures that are used in Stage 2. A renderable texture is one that can hold the color output from a pass. It has different properties attached to it like size, data format, whether it is persistent etc. For more information refer to the renderable textures keyword in chapter 3.1.

2.2.5 State Manager Variables List

The user can specify the name and type of variables that can be linked in Stage 2. Uniview's StateManager variables can be changed at runtime either through run-time commands to Uniview Theater[7] or through Uniview Producer[7]. This gives great flexibility in the use of an effect. For
example, using producer, one could interpolate between different values of a parameter (say 
timeStep) and change the animation speed of a certain effect (like a spiral galaxy).

2.3 Stage 2

Stage 2 consists, at the top level, as a list of pass blocks. At runtime each block is drawn in the order 
in which it is specified in the effect node configuration. A pass block can be further subdivided into 
two pass stages (similar to the parent hierarchy): PassStage1 and PassStage2. PassStage 1 sets the 
pass properties that will be used for PassStage 2. PassStage2 are a series of Material Filter Blocks.

2.3.1 Various Configuration Parameters for Stage 2

The user can specify whether the particular pass is enabled or not. This is a useful debug feature 
while developing an effect. Also the user can scale all the data objects drawn in the particular pass 
uniformly.

2.3.2 Render Target Texture Selection

The user can choose to direct the output of the pass into a renderable texture (those that are 
specified in Stage 1). In the absence of any specification the output goes directly to the frame 
buffer. Render to texture enables many possibilities: like image filtering techniques, shadow maps, 
updating particle systems on the GPU etc.

2.3.3 3D Data List

The user needs to specify what data objects will be drawn in the pass. There can be multiple objects 
(each is a reference to ones specified in Stage 1), and the user can also select an object to be drawn 
multiple times. This provides a means to instance the same object which can be differentiated in the 
shader by the uniform variable, objectNo.

2.3.4 Material Filter

2.3.4.1 Various Configuration Parameters for Material Filter

Like the pass, each material filter can also be enabled or disabled. This makes it possible to not 
draw all sub-meshes of an object that uses that particular material filter. Also there are a list of 
shader preprocessor definition directives that can be enabled or disabled. These affect the code path 
of the shader that is compiled. This is described in more detail in section 3.1.

2.3.4.2 Shader Selection

By default, if no shaders are specified, then the default shader is applied, but one can always specify 
a particular shader(s) for a particular material(s). Currently you can specify a vertex, fragment and a 
geometry shader.
2.3.4.3 Shader Parameter Settings

The shader parameters are the uniform variables that can be specified to the shader. Other than the custom parameters specified by the user there are default parameters which are passed to the shader (if used in the shader) based on the values obtained for the particular material(s) from the 3D data. They can always be overwritten by the user in the configuration script.

2.3.4.4 GL State Settings

Various OpenGL rendering states are exposed to the user. This gives the flexibility of setting up the right GL state for a particular material. For example a transparent material would need the blending state turned on and the depth state turned off.
3 USES Script Lexicon

This chapter describes the USES interface in detail, outlining all keywords used and summarizing their purpose. For clarity the keywords are coloured coded according to the effect node stage (first shown in Figure 2). Section 3.1 describes the USES configuration script keywords. Section 3.2 details in depth the various aspects of the default shader. The default shader is used for any material for which no custom shader is supplied in the configuration script. In Section 3.3 I describe the default system parameters that can be accessed by a shader. These parameters come either from the material of the sub-mesh or are Uniview specific (like the simulation time etc). Section 3.4 describes how to set up different GLSL shader version support in the shader code and Section 3.5 presents some runtime commands to Uniview related to USES (forexample how to change current effect node on a USES module).

3.1 Keywords

mesh
marks the beginning of the USES configuration script

data
usage: data car ./modules/car/car.3ds

gives example path of the 3d model and the name to be used in the script

radius
Default Value: 0.0 (which means the radius would be determined by the model data itself)
specifies the radius of the model in the unit of scene in which it is placed.

enableTypeCheck
Default Value: true
when enabled the system checks the type of the specified parameters in the script with their corresponding uniform variables in the shader. In case of a mismatch the user is informed and the parameters are disabled (that is the script values have no effect on the shader values).

glsIVersion
Default Value 120
usage: glsIVersion 150
sets the GLSL compiler version for the shaders in the effect node. If the specified version is not supported then the closest supported version to that is selected.
Supported values are: 120, 130, 140, 150, 330, 400, 410 and 420.

shadowScale
Default Value: 1.0
usage: shadowScale 0.5
Controls the shadow coverage. The shadow will cover the whole radius of the object and this can be scaled using the value supplied here. To narrow it down, a lower value than 1.0 is used, while to widen it, higher values than 1.0 are used. This should be tweaked when you want to optimize the quality of the shadow, too low a scale will result in some parts of the object (parts that lie outside the scaled radius) displaying artifacts.

renderTexture
usage:
renderTexture
Describes a renderable texture with a name “bumpTexture” of size textureWidth x textureHeight. This texture can later be bound as a target texture for a pass output. The numTextures parameter specifies how many output texture layers this renderable texture supports (This is useful for MRT shaders). “isPersistent” specifies whether the same texture will be available in the next pass with values of the current pass. This is useful for doing a variety of things, such as simulations of particles, because the system can now have memory. “isPingPong” allows the user to read the renderable texture in a pass and write to it in the same pass. Internally this is handled by a pair of swapping renderable textures. For the internal data representation of the renderable textures the following formats are supported: GL_RGBA8, GL_RGB8, GL_RGB32F, GL_RGBA32F, GL_R16F and GL_I32F. “magnify” and “minify” are the texture filter modes. Their values are the GL texture filter constants. All parameters are optional and the defaults values are as follows:

```
name: DefaultRenderTargetTexture
numTextures: 1
textureWidth: 512
textureHeight: 512
isPersistent: false
isPingPong: true
internalTextureFormat: GL_RGBA8
magnify: GL_LINEAR
minify: GL_LINEAR
```
Controls the animation properties of objects (that have skinning or geometric animations in their models). ‘ObjectName1’ is an example name of the object that should match one specified in the data mesh list given through the keyword [data]. ‘seq’ is a keyword that describes which animation sequence of the object is going to be customized. ‘playFrom’ and ‘playTill’ are the starting and stopping time of the sequence. Note that the total time of the animation sequence is given in the model and the user is responsible for knowing that. ‘timeStep’ multiplied by the animation time unit (which is, by default, one second) gives the timestep of the animation.

The default behavior of any animation is to loop through the first animation sequence in the model by the timestep provided in the model itself. By overwriting this using the animation keyword the user can fine-tune the animation speed and select the portion of the animation to be “looped over”.

Note: the sequence to be played in the animation of an object is, by default, the first sequence but can be overwritten by the last sequence specified in the USES configuration script. In the example above, for objectName1, the playing sequence would be sequence 1.

This defines Uniview's statemanager property collections. In the example above two statemanager properties are defined: “Effect1”, “Effect2” each having one single variable property. These, in conjunction with using stateManagerVar parameter, provide a powerful mechanism for runtime customization of effect or control through Producer.

So far the following type of statemanager properties and their keyword are supported:
<table>
<thead>
<tr>
<th>Effect Node variables</th>
<th>Corresponding State Manager Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>vec1f</td>
<td>float</td>
</tr>
<tr>
<td>vec2f</td>
<td>sgVec2</td>
</tr>
<tr>
<td>vec3f</td>
<td>sgVec3</td>
</tr>
<tr>
<td>vec4f</td>
<td>sgVec4</td>
</tr>
</tbody>
</table>

*Table 1: Uniview's Statemanager property types that can be changed through effect nodes*

Note: If the property collection is already defined then the individual properties are appended or updated (if the individual property is already defined as well).

**pass**

starts the pass specifications

**passEnable** *Default Value: true*

*usage: passEnable false*

If the pass is true, the rendering will take place as defined by the pass or else it will be turned off. However, a ‘false’ value doesn’t mean that the pass will not be processed at the time of loading (meaning resources for the pass will be allocated)

If the keyword passEnable is not specified then, by default, the pass is enabled.

**useDataObject**

*usage: useDataObject <NameOfDataObject1> | <count1> <NameOfDataObject2> <count2> <NameOfDataObject3> <count3> ...

Specifies the list of models that are drawn in the corresponding pass. The count specifies how many times the model should be drawn. This, with the help of the default uniform variable “objectNo” in the shader, can be used to draw the object at different places.

**shadowBias**

Default Value: 0.0

*usage: shadowBias 0.01*

Applies a bias (in depth units), where 1.0 means the whole extent of the shadow coverage area. This is used to remove common shadow acne artifacts by adding this bias to the depth comparison in the shadow receiving rendering pass.

**passScale** *Default Value: 1*

*usage: passScale 0.5*

All the data objects are automatically scaled to fit the sphere defined by the ‘radius’ parameter given in their parent scene. However to scale each pass differently 'passScale' can subsequently be used.

**renderTarget**

*usage:
renderTarget
{
    name bumpTexture
    enableDepthClear true
    enableColorClear false
}

If specified then the pass rendering would be to the texture specified by the name keyword (that should match with a renderable texture name given by the renderTexture keyword) instead of to the screen. The ‘enableDepthClear’ controls whether the depth buffer for the render target should be cleared or not and the ‘enableColorClear’ controls whether the color buffer for the render target should be cleared or not. This is useful to control when performing multiple passes but with the same render target. Default values of the parameters, in the absence of specification in the script, are:
name: DefaultRenderTargetTexture
enableDepthClear: true
enableColorClear: true

**shader** Default Behavior: Loads the default shader with default parameters
starts the shader specifications

**type** Must be used if the 'shader' keyword is used
currently only “defaultMeshShader” is the default (and only valid) value for this keyword

**mtrl** Default Value: defaultMeshMaterial
usage: mtrl mtrlName1 | ( mtrlName2 mtrlName 3 ... )
Specifies the name of the material (can be viewed from the open asset import view utility [4]) for which this material filter will be used. If specified then the material is assumed to be defaultMeshMaterial

As the example usage illustrates, you can specify a list of material names for those to which this material filter would be applied. This avoids the need to have redundant material filters in the same pass with the only difference being the mtrl name.

**vertexShader** Default Value: ./data/shaders/meshShader.vs
usage: vertexShader <file-path+name>

**fragmentShader** Default Value: ./data/shaders/meshShader.fs
usage: fragmentShader <file-path+name>

**geometryShader** Default Value: (empty string)
usage: geometryShader <file-path+name>

**enable** Default Value: true
With this all submeshes of objects that use this particular material filter can be disabled (i.e. not rendered)

**texFlipU** Default Value: false

19
Flips the U texture coordinate if 'true' is specified.¹

**texFlipV** Default Value: false
Flips the U texture coordinate if 'true' is specified²

**define**
usage: define CONSTANT_NAME

By default all the necessary constants will be defined for a material depending on the texture available for the model or by having tangents or binormals or by having skeletal animation or by the glsl version supported by the system. But, for example, to define an additional type of texture (not in the model e.g, SG_NORMAL_TEXTURE for bump mapping) we have to use the define construct.

For a working example of these keywords look at the default shader (given in the appendix in Section 7.1). The name of the constant corresponds to their counterpart in the “default” glsl shader used in the files: “meshShader.vs” and “meshShader.fs” (the vertex and fragment shaders respectively given in the appendix of this document in Section 7.1.1 and 7.1.2).

These defines are automatically defined for each shader (custom or default) depending on the material of the mesh.

<table>
<thead>
<tr>
<th>Defined constants used in the shader</th>
</tr>
</thead>
<tbody>
<tr>
<td>UV_DIFFUSE_TEXTURE</td>
</tr>
<tr>
<td>UV_AMBIENT_TEXTURE</td>
</tr>
<tr>
<td>UV_SPECULAR_TEXTURE</td>
</tr>
<tr>
<td>UV_NORMAL_TEXTURE</td>
</tr>
<tr>
<td>UV_EMISSIVE_TEXTURE</td>
</tr>
<tr>
<td>UV_SHININESS_TEXTURE</td>
</tr>
<tr>
<td>UV_LIGHTMAP_TEXTURE</td>
</tr>
<tr>
<td>UV_DISPLACEMENT_TEXTURE</td>
</tr>
<tr>
<td>UV_REFLECTION_TEXTURE</td>
</tr>
<tr>
<td>UV_OPACITY_TEXTURE</td>
</tr>
<tr>
<td>UV_OPACITY_TEXTURE_REGISTER_MASK_R</td>
</tr>
<tr>
<td>UV_OPACITY_TEXTURE_REGISTER_MASK_A</td>
</tr>
<tr>
<td>UV_MESH_HAS_TANGENTS</td>
</tr>
<tr>
<td>UV_MESH_HAS_BINORMALS</td>
</tr>
<tr>
<td>UV_SPECULAR_MATERIAL</td>
</tr>
<tr>
<td>UV_DEFINE_SKELETAL_ANIMATION</td>
</tr>
</tbody>
</table>

¹Only applicable for default shader
²Only applicable for default shader
Table 2: Shader constants that are automatically defined for a shader based on the material of a sub mesh

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UV_DEFINE_GLSL_VERSION_1_20</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_1_30</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_1_40</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_1_50</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_3_30</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_4_00</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_4_10</td>
<td></td>
</tr>
<tr>
<td>UV_DEFINE_GLSL_VERSION_4_20</td>
<td></td>
</tr>
</tbody>
</table>

**undefined**

*usage: CONSTANT_NAME*

uses the same constants as ‘define’ but, instead, makes sure that the particular constant is not defined for the custom or default shader for that material filter. Furthermore for the default shader it would not call the particular code that is related to the define block (as can be seen from the default shader listed in the appendix). This keyword can be useful for debugging purposes.

**texture**

*usage: texture VarName PathOfFile*

```glsl
{
    # optional properties
    wrapModeS   GL_REPEAT
    wrapModeR   GL_REPEAT
    wrapModeT   GL_REPEAT
    magnify     GL_NEAREST
    minify      GL_NEAREST
}
```

Specifies the name of the uniform texture sampler used in the glsl shader and the path of the texture in the file system and “optional” texture access and filtering modes. The type of texture sampler is decided on the type of texture specified. All supported image formats are understood to be sampler2D in the shader except for .dds extension. In case of .dds texture specified: it can be a sampler2D, sampler3D ( a 3d texture ) or a samplerCube (for cube maps). Again the type of the texture specified should match the type in the shader.

The ‘wrapMode’ can take values for the access properties for OpenGl textures. Default: GL_CLAMP_TO_EDGE. And the ‘magnify’/ ‘minify’ take OpenGl texture filtering mode values. Default: GL_LINEAR

Note: Regardless of setting the optional properties, the lines following the ‘texture’ command should have open and close brackets: ‘{‘ and the next one ‘}’

In case the shader is the default shader and the name of the texture is one of the default texture
names (e.g. diffuseTexture) of textures that are specified in the model format, then they are overwritten. This way we can override the default texture behavior of the 3D model.

**textureFBO**

usage: 
```
textureFBO Tex bumpTexture | <texture number>
```

Like the ‘texture’ keyword ‘textureFBO’ specifies the name of the uniform texture samples used in the glsl shader. But instead of a path it uses the renderable texture name which should be specified in the configuration file using the keyword: `renderTexture`. The `<texture number>` option is optional. It is used if the textureFBO is a multitexture. By default the value is 0 (meaning the first texture).

**sceneTransformVar**

Transforms a position vector from one scene to another scene.

usage: 
```
SceneTransformVar DstSceneName SourceSceneName VarNameInShader x y z
```

The VarNameInShader should correspond with the VarNameInShader vec3 variable in the shader. Furthermore, the scene should be loaded prior to the script being called (hence the DstSceneName and SourceSceneName should precede the current module in the autorun.conf)

**stateManagerVar**

Links a variable in shader with a property in the state manager. Now whenever the value in the statemanager is changed the shader variable value is updated accordingly. The datatype of the two variables should match, otherwise an error is issued.

usage: 
```
ModuleNameInStateManager.PropertyName VarNameInShader
```

Just like ‘SceneTransformVar’ the statemanager property should be defined prior to the call of this particular script.

**glState**

corresponds to setting certain states of opengl for the rendering of every sub-mesh that falls in the particular material filter.

The possible values (and an example usage) and their corresponding opengl calls are given in the table below. For brevity’s sake, the parameters number and type are not given. That information can easily be obtained by referring to the particular opengl function as the value and the order of the parameters that should be used in the script are the same as the parameter to the OpenGL function calls.³

<table>
<thead>
<tr>
<th>Script Command for glState</th>
<th>Corresponding OpenGL Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>UV_BLEND_COLOR</td>
<td>glBlendColor(...)</td>
</tr>
<tr>
<td>UV_BLEND_ENABLE</td>
<td>glEnable()/glDisable() with GL_BLEND</td>
</tr>
<tr>
<td>UV_BLEND_EQUATION</td>
<td>glBlendEquation(...)</td>
</tr>
<tr>
<td>UV_BLEND_EQUATION_SEPARATE</td>
<td>glBlendEquationSeparate(...)</td>
</tr>
<tr>
<td>UV_BLEND_FUNC</td>
<td>glBlendFunc(...)</td>
</tr>
</tbody>
</table>

³ except for enable/disabling commands where the values ‘true’ and ‘false’ should be used
<table>
<thead>
<tr>
<th>UV_BLEND_FUNC_SEPARATE</th>
<th>glBlendFuncSeparate(...)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UV_CLEAR_COLOR</td>
<td>glColor4f(...)</td>
</tr>
<tr>
<td>UV_COLOR_4F</td>
<td>glClearColor(...)</td>
</tr>
<tr>
<td>UV_CLEAR_DEPTH</td>
<td>glClearDepth(...)</td>
</tr>
<tr>
<td>UV_CLEAR_STENCIL</td>
<td>glClearStencil(...)</td>
</tr>
<tr>
<td>UV_CULL_FACE_ENABLE</td>
<td>glEnable/glDisable with GL_CULL_FACE</td>
</tr>
<tr>
<td>UV_CULL_MODE</td>
<td>glCullFace(...)</td>
</tr>
<tr>
<td>UV_DEPTH_ENABLE</td>
<td>glEnable/glDisable with GL_DEPTH_TEST</td>
</tr>
<tr>
<td>UV_DEPTH_FUNCTION</td>
<td>glDepthFunc(...)</td>
</tr>
<tr>
<td>UV_DEPTH_RANGE</td>
<td>glDepthRange(...)</td>
</tr>
<tr>
<td>UV_FRONT_FACE</td>
<td>glFrontFace(...)</td>
</tr>
<tr>
<td>UV_LINE_WIDTH</td>
<td>glLineWidth(...)</td>
</tr>
<tr>
<td>UV_POLY_BACK_MODE</td>
<td>glPolygonMode(GL_BACK,...)</td>
</tr>
<tr>
<td>UV_POLY_FRONT_AND_BACK_MODE</td>
<td>glPolygonMode(GL_FRONT_AND_BACK,...)</td>
</tr>
<tr>
<td>UV_POLY_FRONT_MODE</td>
<td>glPolygonMode(GL_FRONT,...)</td>
</tr>
<tr>
<td>UV_POLYOFFSET</td>
<td>glPolygonOffset(...)</td>
</tr>
<tr>
<td>UV_POLYOFFSET_FILL_ENABLE</td>
<td>glEnable/glDisable with GL_POLYGON_OFFSET_FILL</td>
</tr>
<tr>
<td>UV_SCISSOR</td>
<td>glScissor(...)</td>
</tr>
<tr>
<td>UV_SCISSOR_ENABLE</td>
<td>glEnable/glDisable with GL_SCISSOR_TEST</td>
</tr>
<tr>
<td>UV_STENCIL_FUNCTION</td>
<td>glStencilFunc(...)</td>
</tr>
<tr>
<td>UV_STENCIL_ENABLE</td>
<td>glEnable/glDisable with GL_STENCIL_TEST</td>
</tr>
<tr>
<td>UV_STENCIL_FUNC_SEPARATE_BACK</td>
<td>glStencilFuncSeparate(GL_BACK,...)</td>
</tr>
<tr>
<td>UV_STENCIL_FUNC_SEPARATE_BOTH</td>
<td>glStencilFuncSeparate(GL_FRONT_AND_BACK,...)</td>
</tr>
<tr>
<td>UV_STENCIL_FUNC_SEPARATE_FRONT</td>
<td>glStencilFuncSeparate(GL_FRONT,...)</td>
</tr>
<tr>
<td>UV_STENCIL_OP</td>
<td>glStencilOp(...)</td>
</tr>
<tr>
<td>UV_STENCIL_OP_SEPARATE_BACK</td>
<td>glStencilOpSeparate(GL_BACK,...)</td>
</tr>
<tr>
<td>UV_STENCIL_OP_SEPARATE_BOTH</td>
<td>glStencilOpSeparate(GL_FRONT_AND_BACK,...)</td>
</tr>
<tr>
<td>Parameter</td>
<td>GL Function</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>UV_STENCIL_OP_SEPARATE_FRONT</td>
<td>glStencilOpSeparate(GL_FRONT,...)</td>
</tr>
<tr>
<td>UV_STENCIL_VALUE_MASK</td>
<td>glStencilMask(...)</td>
</tr>
<tr>
<td>UV_STENCIL_VALUE_MASK_SEPARATE_BACK</td>
<td>glStencilMaskSeparate(GL_BACK,...)</td>
</tr>
<tr>
<td>UV_STENCIL_VALUE_MASK_SEPARATE_BACK</td>
<td>glStencilMaskSeparate(GL_FRONT_AND_BACK,...)</td>
</tr>
<tr>
<td>UV_STENCIL_VALUE_MASK_SEPARATE_FRONT</td>
<td>glStencilMaskSeparate(GL_FRONT,...)</td>
</tr>
<tr>
<td>UV_TEXTURE_3D_ENABLE</td>
<td>glEnable/glDisable with GL_TEXTURE_3D</td>
</tr>
<tr>
<td>UV_TEXTURE_CUBE_MAP_EXT_ENABLE</td>
<td>glEnable/glDisable with GL_TEXTURE_CUBE_MAP_EXT</td>
</tr>
<tr>
<td>UV_WRITE_MASK_COLOR</td>
<td>glColorMask(...)</td>
</tr>
<tr>
<td>UV_WRITE_MASK_DEPTH</td>
<td>glDepthMask(...)</td>
</tr>
</tbody>
</table>

Table 3: Configurable GL States in USES

3.1.1 Uniform Parameters

Following are the uniform parameters that correspond one-on-one with the uniform variables in the shader

- **parameter1f** - corresponds to “uniform float” in glsl
  usage: parameter1f VarName 1.0

- **parameter1i** - corresponds to “uniform bool” or “uniform int” in glsl
  usage: parameter1i VarName 1

- **parameter1fv** - corresponds to “uniform float[]” in glsl
  usage: parameter1fv VarName 3 0.4 0.1 1.0
  where '3' is the size of array (and must match the corresponding size in the shader)

- **parameter1iv** - corresponds to “uniform int[]” or “uniform bool[]” in glsl
  usage: parameter1iv VarName 3 1 0 1
  where '3' is the size of array (and must match the corresponding size in the shader)

- **parameter2f** - corresponds to “uniform vec3” in glsl
  usage: parameter2f VarName 3 10

- **parameter3f**
  usage: parameter3f VarName 1.0 0.3 0.4

- **parameter4f**
usage: parameter4f VarName 0.3 4.0 4.3 1.0

parameter[234]fv
usage: parameter3fv VarName
    {  
        0 0.2 0.4  
        0 3.3 0.4  
        0.5 0.2 0.4  
    }

parameter2x2f
parameter3x3f
parameter4x4f
These are nxn matrices corresponding to “uniform matn” in glsl
where n = 2,3,4

usage: parameter4x4f matrix
    {  
        1 0 0 0  
        0 1 0 0  
        0 0 1 0  
        0 0 0 1  
    }
The brackets and each row is on its own column. The matrix is specified in column-major order
similar to opengl.

3.2 The Default Shader

The default behavior of the default shader if no modifications are applied in the configuration file is
to render the model as a phong shaded material. The default values of the material are as such:
diffuseMtrl (0.7,0.7,0.7)
ambientMtrl (0.0 0.0 0.0)
specularMtrl (1.0 1.0 1.0)
emissiveMtrl (0.0 0.0 0.0)
shininessMtrl 15
opacityMtrl 1.0

These values are overwritten based on what values are present in the model. They can be
overwritten further by the user in the USES configuration file.

The ‘define’ construct are enabled depending on the presence of the various things like the kind of
textures available in the material. So if the model has a diffuse texture and a normal texture the two
constants defined would be: UV_DIFFUSE_TEXTURE and UV_NORMAL_TEXTURE

3.2.1 Default Parameters in Default and Custom Shaders

---

4 The shader is listed in the appendix
The following are parameters loaded from the 3D model used in the default mesh shader and, if used in custom shader, would be linked accordingly. Their values can be overwritten through the configuration script.

### 3.2.2 Textures
These are textures corresponding to “uniform sampler2D” in glsl and can be modified with the keyword “texture”

<table>
<thead>
<tr>
<th>Texture Sample Names in GLSL</th>
</tr>
</thead>
<tbody>
<tr>
<td>diffuseTexture</td>
</tr>
<tr>
<td>ambientTexture</td>
</tr>
<tr>
<td>specularTexture</td>
</tr>
<tr>
<td>normalTexture</td>
</tr>
<tr>
<td>emissiveTexture</td>
</tr>
<tr>
<td>shininessTexture</td>
</tr>
<tr>
<td>lightmapTexture</td>
</tr>
<tr>
<td>displacementTexture</td>
</tr>
<tr>
<td>reflectionTexture</td>
</tr>
<tr>
<td>opacityTexture</td>
</tr>
</tbody>
</table>

*Table 4: Available textures in the shader based on the material of the sub mesh*

### 3.2.3 Texture Modifiers
The strength value is multiplied with the texture value
The op (operation) selects how to blend the texture value with base color

<table>
<thead>
<tr>
<th>Texture Modifier Names</th>
<th>Type of the parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>diffuseTexStrength</td>
<td>parameter1f</td>
</tr>
<tr>
<td>diffuseTexOp</td>
<td>parameter1i</td>
</tr>
<tr>
<td>ambientTexStrength</td>
<td>parameter1f</td>
</tr>
<tr>
<td>ambientTexOp</td>
<td>parameter1i</td>
</tr>
<tr>
<td>specularTexStrength</td>
<td>parameter1f</td>
</tr>
<tr>
<td>specularTexOp</td>
<td>parameter1i</td>
</tr>
<tr>
<td>emissiveTexStrength</td>
<td>parameter1f</td>
</tr>
<tr>
<td>lightmapTexStrength</td>
<td>parameter1f</td>
</tr>
<tr>
<td>opacityTexStrength</td>
<td>parameter1f</td>
</tr>
</tbody>
</table>
Table 5: Available texture modifiers in the shader based on the material of the sub mesh

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Type of the Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflectionTexStrength</td>
<td>parameter1f</td>
</tr>
<tr>
<td>reflectionTexOp</td>
<td>parameter1i</td>
</tr>
</tbody>
</table>

3.2.4 Material

<table>
<thead>
<tr>
<th>Material Names</th>
<th>Type of the Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>diffuseMtrl</td>
<td>parameter3f</td>
</tr>
<tr>
<td>ambientMtrl</td>
<td>parameter3f</td>
</tr>
<tr>
<td>specularMtrl</td>
<td>parameter3f</td>
</tr>
<tr>
<td>shininessMtrl</td>
<td>parameter1f</td>
</tr>
<tr>
<td>emissiveMtrl</td>
<td>parameter3f</td>
</tr>
<tr>
<td>opacityMtrl</td>
<td>parameter3f</td>
</tr>
</tbody>
</table>

Table 6: Available material parameters in the shader based on the material of the sub mesh

3.3 Default System Parameters

These are parameters loaded from the system at each draw call and normally shouldn't be overwritten in the configuration file (unless for debugging purposes). These parameters can be used by any custom shader because the system would bind these variables if they are found in the shader.

3.3.1 Attributes

**vertexAttrib**
vec3- vertex attributes loaded from the model

**normalAttrib**
vec3- normal attributes loaded from the model

**texCoordAttrib0**
vec2- texture coordinates loaded from the model

**tangentAttrib** and **bitangentAttrib**
vec3- attributes loaded from the 3D model if they are specified in the model.

**weights**
vec4 - if the model has animation then weights would have blending weights of four vertices.
**boneIndices**
vec4 - if the models has animation then boneIndices would have blending indices of four vertices

### 3.3.2 Uniforms

**lightPos**
vec4 - light position in object model space

**randomVec4**
**randomVec3**
**randomVec2**
**randomFloat**

**time**
float - floating point system time

**passNo**
int - the pass number

**objectNo**
int - the object number: which is the counted from the object list specified by the keyword `useDataObject <list>`
e.g: useDataObject obj1 1 obj2 20 obj3 50
Then the value of the parameter would go from 0 to 70 (as there are 71 objects)

**meshNo**
int - mesh number based on total sub-meshes in the object.

**cameraPos**
vec4 - eye position in *scene* coordinate space (where scene correspond to scene of the module). cameraPos will always be in that scene even if the camera is moved to another scene. To get it say in the object space: use `scene2ObjectMatrix`

**object2SceneMatrix**
mat4 - transform to convert from object space to scene (the scene in which the module was specified) space

**scene2ObjectMatrix**
mat4 - transform to convert from scene (the scene in which the module was specified) space to object space

**boneMatrices**
mat4 [] - If the object has animations then boneMatrices is a mat4 array.

---

5Currently this vec4 has actually direction of the light source in the model space given in its xyz components
The following are equivalent to their respective built-in ‘gl_variablename’ counterpart.

\[ \text{modelViewMatrix} \quad \text{mat4} \]

\[ \text{projectionMatrix} \quad \text{mat4} \]

\[ \text{modelViewProjectionMatrix} \quad \text{mat4} \]

\[ \text{normalMatrix} \quad \text{mat4} \]

### 3.4 GLSL Version Support

Depending on the GLSL version support of the running system, following constants will be defined for each shader.

- \( \text{UV\_GLSL\_VERSION\_1\_20} \)
- \( \text{UV\_GLSL\_VERSION\_1\_30} \)
- \( \text{UV\_GLSL\_VERSION\_1\_40} \)
- \( \text{UV\_GLSL\_VERSION\_1\_50} \)
- \( \text{UV\_GLSL\_VERSION\_3\_30} \)
- \( \text{UV\_GLSL\_VERSION\_4\_00} \)
- \( \text{UV\_GLSL\_VERSION\_4\_10} \)
- \( \text{UV\_GLSL\_VERSION\_4\_20} \)

*example usage:* placing the following code above any shader (fragment or vertex) file

```c
#ifdef UV_GLSL_VERSION_1_50
// version 150 specific code
#elif UV_GLSL_VERSION_1_20
// version 120 specific code
#endif
```

### 3.5 Run Time Syntax

- \(<\text{module name}>\).reload
  - reloads the USES configuration script/s.

- \(<\text{module name}>\).mesh reloadshaders
  - rebuilds all the shaders attached to the USES object

- \(<\text{module name}>\).mesh debugDraw
  - Toggles between the regular drawing and debug drawing for USES. The debug draw is the individual passes output to the screen in different viewports.
<module name>.mesh debugGridSize x y
By default there are 3x3 viewports for the debug draw.

<module name>.mesh activeeffect x
If USES object has more than one effect node attached then this changes between different effect
nodes. The indexing starts from 0.

system.reloadallshaders
Reloading all the shaders of all USES modules.
4 Results

The results in the section are courtesy of SCISS AB that were generated during USES workshop with prominent users of Uniview held in the first week of May, 2012. All effects are in real time with frame-rates well above 60 fps using current to 3 years old Nvidia's graphics cards. In Figure 5 a rainbow is generated over the Earth by raytracing a quad in the pixel shader. Figure 4 illustrates the effect node configuration script for the effect. Figure 6 shows an imported mesh module of Hubble in 3ds format orbiting around the earth. It is a two pass effect with the first pass generating the shadow map from the light direction. Figure 7 shows the same object but this time in Earth's shadow. Notice the bluish glow resulting from the Earth. Figure 8 shows a detailed ISS model (courtesy Toshiyuki Takahei). The effect has shadows, bump maps, diffuse and specular textures. Also the bluish glow from the earth is noticeable on the shadowy sides.

Figure 9, Figure 10 and Figure 11 show different views of visualization of earthquakes around the globe. Figure 12 illustrates the effect node configuration script. Notice the use of property collection to customize the effect during runtime. Figure 14 and Figure 15 show a visualization of asteroid data sets orbiting around the solar system. There are a total of about 580,000 asteroids (depicted as fading lines through the geometry shader). A single asteroid is defined by a set of 7 values that are together called the orbital elements. Figure 13 graphically showcases the orbital elements and Figure 16 gives the asteroid effect's configuration script. Lastly Figure 17 is a funky effect (by changing aspect ratio in uniview and some other parameters) to show that USES can also have fun in its use.

```
DISCLAIMER: The information is the intellectual property of SCISS AB

mesh
{
  data quad ./modules/rainbow/quad.3ds
  # use a radius to avoid frustum culling
cullRadius 1000
  glslVersion 150
  pass
  {
    passEnable true
    useDataObject quad
    shader
    {
      type defaultMeshShader
      {
        vertexShader ./modules/rainbow/vertex_shader.glsl
        fragmentShader ./modules/rainbow/fragment_shader.glsl
        parameter1f alpha 1.0
        glState
        {
          UV_CULL_FACE_ENABLE false
          UV_DEPTH_ENABLE false
          UV_BLEND_ENABLE true
          UV_BLEND_FUNC GL_ONE GL_ONE
        }
      }
    }
  }
}
```
Figure 5: A Rainbow Effect on Earth

Figure 6: Shadow Mapped Hubble Model

Figure 7: Earth Glow on Hubble
Figure 8: Shadow Mapped ISS Model with Earth Glow

Figure 9: Earthquake Data Visualization 1
DISCLAIMER: The information is the intellectual property of SCISS AB

```json
mesh
{
  data dataMesh1 /modules/earthquakes/models/ge1.quakes.raw
  gisVersion 330
  creaseAngle 180.0
  propertyCollection
  {
    quakes
    {
      vec4f colorShallow 1.0 0.8 0.2 0.5
      vec4f colorDeep 0.5 0.1 0.3 0.8
      vec4f reveal 1 1 0
      vec4f preReveal 0
      vec4f postReveal 0.1
      vec4f smoothReveal 0
    }
  }
  pass
  {
    useDataObject dataMesh1
  shader
  {
    type defaultMeshShader
    {
      vertexShader /modules/earthquakes/meshPass0.vs
      fragmentShader /modules/earthquakes/meshPass0.fs
      geometryShader /modules/earthquakes/meshPass0.gs
      StateManager/Var quakes.colorShallow colorShallow
      StateManager/Var quakes.colorDeep colorDeep
      StateManager/Var quakes.reveal reveal
      StateManager/Var quakes.preReveal preReveal
      StateManager/Var quakes.postReveal postReveal
      StateManager/Var quakes.smoothReveal smoothReveal
      parameter/2f depth 6000 6380
      gState
      {
        UV_BLEND_FUNC GL_SRC_ALPHA GL_ONE
        UV_DEPTH_ENABLE false
        UV_BLEND_ENABLE true
      }
    }
  }
}
```

**Figure 12:** Effect Node configuration of an earthquake data visualization

**Figure 13:** An illustration of the orbital elements for the asteroid data set (image courtesy of wikipedia licensed under GFDL)
Figure 14: Asteroids Visualization 1

Figure 15: Asteroid Visualization 2
Figure 16: Effect Node configuration of an asteroid data visualization

Figure 17: Funky effect created by changing aspect ratio in Uniview
5 Conclusions

USES has proven to be a valuable addition to Uniview's arsenal of astronomical visualization tools. It makes the importing of many different 3D formats easy and accessible. Which in turn leads to the possibility of rapid prototyping which is essential to deliver quality features quickly. In addition to just getting the data in, making it look good is also easier thanks to the many customization options that USES provides to the user. Effects like meteorite showers, auroras, etc. could be added using USES as the base engine.

There is, however, still work to be done. In any content pipeline the fastest and most productive methods need to be developed. The results from the USES workshop shown in chapter 4 are a move towards that. With the help from USES users the development of the product would take that route, identifying the best 3D formats and best practices to incorporate various effects as well as building the right tools. Currently USES exposes a script which is created manually, but in the future mesh importing tools (with various effects like self shadowing etc) could be added that generate the script automatically.
Future Work

There is always potential of improvements in the system. Some ideas for future development are:

- Support for tesselation shader stages introduced in later opengl versions.
- Support for vertex transform feedback giving potential for creating even better particle systems.
- Support for more than the standard data streams from the object (like position, normals, textures coordinates, tangents etc) and supporting any custom data item per vertex.
- Accessing Uniview Video playback feature from USES. A powerful feature if a video texture can be used in USES. This could open up a whole new level of visualization, faster prototyping (incase the video is a simulation like animating normals/displacement maps etc).
- A proper graphical user interface for USES. This would increase the productivity of the users by not configuring every thing in scripts.
6 References

7 Appendix

7.1 Default Shader

7.1.1 Vertex Shader

```glsl
varying vec3 lightDir, normal, eyeVec;
varying vec2 texCoord0;

#ifdef UV_NORMAL_TEXTURE
attribute vec3 tangentAttrib,bitangentAttrib;
#endif

#ifdef UV_DEFINE_SKELETAL_ANIMATION
uniform mat4 boneMatrices[64];
attribute vec4 weights;
attribute vec4 boneIndices;
#endif

attribute vec3 vertexAttrib;
attribute vec3 normalAttrib;
attribute vec2 texCoordAttrib0;

uniform vec4 lightPos;
uniform float time;
uniform bool texFlipU;
uniform bool texFlipV;

uniform mat4 modelViewMatrix;
uniform mat4 modelViewProjectionMatrix;
uniform mat4 normalMatrix;

varying vec4 temp;

void main(void)
{

#ifdef UV_DEFINE_SKELETAL_ANIMATION
    vec4 newVertex = vec4(0,0,0,0);
    vec4 newNormal = vec4(0,0,0,0);
    int index;

    for (int i=0; i<4; i++)
    {
        index = int(boneIndices[i]);
        newVertex += boneMatrices[index] * vec4(vertexAttrib,1.0) * weights[i];
        newNormal += boneMatrices[index] * vec4(normalAttrib,0.0) * weights[i];
    }

    normal = (normalMatrix * vec4(newNormal.xyz,0.0)).xyz;
    gl_Position = modelViewProjectionMatrix* vec4(newVertex.xyz,1.0);

```
#else
   normal = (normalMatrix * vec4(normalAttrib,0.0)).xyz;
   gl_Position = modelViewProjectionMatrix* vec4(vertexAttrib,1.0);
#endif

vec3 vVertex = vec3(modelViewMatrix* vec4(vertexAttrib,1.0));
vec3 tmpVec = normalize( (modelViewMatrix* lightPos).xyz );

#ifndef UV_NORMAL_TEXTURE
   lightDir = tmpVec;
   eyeVec = -vVertex;
#else
   vec3 n = normalize( (normalMatrix * vec4(normalAttrib,0.0)).xyz );

   #ifdef UV_MESH_HAS_TANGENTS
   vec3 t = normalize( (normalMatrix  * vec4(tangentAttrib,0.0)).xyz);
   #else
   vec3 t = cross(n,vec3(0,1,0) );
   #endif

   #ifdef UV_MESH_HAS_BINORMALS
   vec3 b = normalize( (normalMatrix  * vec4(bitangentAttrib,0.0)).xyz);
   #else
   vec3 b = cross(n,t);
   #endif

   lightDir.x = dot(tmpVec,t);
   lightDir.y = dot(tmpVec,b);
   lightDir.z = dot(tmpVec,n);
   tmpVec = -vVertex;
   eyeVec.x = dot(tmpVec, t);
   eyeVec.y = dot(tmpVec, b);
   eyeVec.z = dot(tmpVec, n);
#endif

if (texFlipU)
   texCoord0.s  = 1 - texCoordAttrib0.s;
else
   texCoord0.s  = texCoordAttrib0.s;

if (texFlipV)
   texCoord0.t  = 1 - texCoordAttrib0.t;
else
   texCoord0.t  = texCoordAttrib0.t;
}

42
7.1.2 Fragment Shader

precision highp float;

varying vec3 lightDir, normal, eyeVec;
varying vec2 texCoord0;

uniform float time;

#ifdef UV_DIFFUSE_TEXTURE
uniform sampler2D diffuseTexture;
uniform float diffuseTexStrength;
uniform int diffuseTexOp;
#endif

#ifdef UV_AMBIENT_TEXTURE
uniform sampler2D ambientTexture;
uniform float ambientTexStrength;
uniform int ambientTexOp;
#endif

#ifdef UV_SPECULAR_TEXTURE
uniform sampler2D specularTexture;
uniform float specularTexStrength;
uniform int specularTexOp;
#endif

#ifdef UV_NORMAL_TEXTURE
uniform sampler2D normalTexture;
#endif

#ifdef UV_EMISSIVE_TEXTURE
uniform sampler2D emissiveTexture;
uniform float emissiveTexStrength;
#endif

#ifdef UV_LIGHTMAP_TEXTURE
uniform sampler2D lightmapTexture;
uniform float lightmapTexStrength;
#endif

#ifdef UV_OPACITY_TEXTURE
uniform sampler2D opacityTexture;
uniform float opacityTexStrength;
#endif

#ifdef UV_REFLECTION_TEXTURE
uniform sampler2D reflectionTexture;
uniform float reflectionTexStrength;
uniform int reflectionTexOp;
#endif

uniform vec3 diffuseMtrl, ambientMtrl, emissiveMtrl;

#ifdef UV_SPECULAR_MATERIAL
43
uniform vec3 specularMtrl;
uniform float shininessMtrl;
#endif

uniform float opacityMtrl;

#ifdef UV_REFLECTION_TEXTURE
vec2 SphereMap(in vec3 ecPosition3, in vec3 normal)
{
    float m;
    vec3 r, u;
    u = normalize(ecPosition3);
    r = reflect(u, normal);
    m = 2.0 * sqrt(r.x * r.x + r.y * r.y + (r.z + 1.0) * (r.z + 1.0));
    return vec2(r.x / m + 0.5, r.y / m + 0.5);
}
#endif

vec4 applyOperation(vec4 T1, vec4 T2, int op)
{
    vec4 T = vec4(0,0,0,0);

    if (op == 0)
        T = T1 * T2;
    else if (op == 1)
        T = T1 + T2;
    else if (op == 2)
        T = T1 - T2;
    else if (op == 3)
    {
        /*
         if (T2.r != 0.0 && T2.g != 0.0 && T2.b != 0.0)
        {
            T.r = T1.r / T2.r;
            T.g = T1.g / T2.g;
            T.b = T1.b / T2.b;
        }
        */
    }
    else if (op == 4)
        T = (T1 + T2) - (T1*T2);
    else
        T = T1 + (T2 - 0.5);

    return T;
}

void main(void)
{
    vec4 finalDiffuseColor = vec4(0.0);
    vec4 finalAmbientColor  = vec4(ambientMtrl,1.0);
    vec4 finalSpecularColor = vec4(0.0);
vec4 finalColor;

#ifdef UV_AMBIENT_TEXTURE
finalAmbientColor = ambientTexStrength * 
applyOperation(finalAmbientColor,texture2D(ambientTexture,texCoord0.st),ambientTexOp);
#endif

#ifdef UV_NORMAL_TEXTURE
vec3 bump = normalize(texture2D(normalTexture,texCoord0.st).rgb * 2.0 - 1.0);
vec3 N = bump;
#else
vec3 N = normalize(normal);
#endif
vec3 L = normalize(lightDir);
float lambertTerm = (dot(N,L));
float specular;
if (lambertTerm > 0.0)
{
    finalDiffuseColor = vec4(diffuseMtrl,1);

    #ifdef UV_DIFFUSE_TEXTURE
    finalDiffuseColor = diffuseTexStrength * 
    applyOperation(finalDiffuseColor,texture2D(diffuseTexture,texCoord0.st),diffuseTexOp);
    #endif

    finalDiffuseColor *= lambertTerm;

    #ifdef UV_SPECULAR_MATERIAL
    vec3 E = normalize(eyeVec);
    vec3 R = reflect(-L,N);
    specular = pow( max(dot(R,E), 0.0) , shininessMtrl );

    finalSpecularColor = vec4(specularMtrl,1);
    #ifdef UV_SPECULAR_TEXTURE
    applyOperation(finalSpecularColor,texture2D(specularTexture,texCoord0.st),specularTexOp);
    #endif
    finalSpecularColor *= specular;
    #endif
}
finalColor.rgb = (finalAmbientColor + finalDiffuseColor + finalSpecularColor).rgb;

#ifdef UV_EMISSIVE_TEXTURE
45
}
finalColor.rgb += emissiveTexStrength * texture2D(emissiveTexture, texCoord0.st).rgb;
#else
finalColor.rgb += emissiveMtrl;
#endif

#ifdef UV_LIGHTMAP_TEXTURE
finalColor.rgb *= texture2D(lightmapTexture, texCoord0.st).rgb * lightmapTexStrength;
#endif

#ifdef UV_REFLECTION_TEXTURE
finalColor = reflectionTexStrength * applyOperation(finalColor,texture2D(reflectionTexture,SphereMap(-eyeVec,N)),reflectionTexOp);
#endif

#ifdef UV_OPACITY_TEXTURE
finalColor.a = opacityTexStrength * texture2D(opacityTexture, texCoord0.st).UV_OPACITY_TEXTURE_REGISTER_MASK;
#else
finalColor.a = opacityMtrl;
#endif

gl_FragColor = finalColor;
}