MGE User Guide

General Guidelines

Asset Placement

All assets should be placed inside the **"assets/mge"** folder. Inside that folder there are different folders that correspond to the different asset types, like textures, models, audio, etc.

Shaders

Shader Types

MGE supports 3 shader types:

- Vertex
- Fragment
- Geometry

Every shader must have a vertex and a fragment program and can **optionally** have a geometry shader. To define a specific program with a specific type, use the following file extensions:

- .vs – vertex shader
- .fs – fragment shader
- .gs – geometry shader

File Placement

All shaders must be placed inside **the "assets/mge/shaders"** folder. Inside that folder there are 2 other folders: **"lit"** and **"unlit"**. Depending on whether the shader applies lighting or not, it should be placed inside one of these folders. All shader program files (vertex, fragment and geometry (if applicable)) **must be placed together in the same folder**.

C++ API

Once a shader is place inside the **shaders** folder, it is loaded and cached automatically at program start.

Finding Shaders

To use a shader, you need to retrieve a pointer to it. To do that, use the **static method** inside the **Shader** class

- **Shader::Find("Shader Name Here")**

Example:

```
Shader* myLitShader = Shader::Find("lit/lit");
```

This code retrieves a pointer to the **lit** shader which is inside the folder **"assets/mge/shaders/lit".**

*The first lit is the folder named lit. The second lit is the name of the shader.

Using Shaders

Binding Shaders

To let OpenGL know that you are using a specific shader to draw things right now, you need to bind the shader. This is done simply by calling the **Bind()** method on the shader.

After you are done drawing, make sure you call **Unbind()** on the shader! If you forget to do so, there won't be any visible problems, but it might cause performance issues.

Example:

```
Shader* myShader = Shader::Find("unlit/basicShader");

myShader->Bind();

… Draw Stuff Here …

myShader->Unbind();
```

Validate Shaders

Validating shaders is a debugging tool useful to verify whether the shader is still in a good state for OpenGL to use it. This is rarely used.

Example:

```
myShader->Validate();
```

If there are some problems, an output to the console will be provided.

Getting Attributes

If you need to work with raw OpenGL and you need the ID of a certain attribute inside the Shader, use the method:

GetAttribute("Attribute Name")

This will return an unsigned GL integer (GLuint) and you can use this number with raw OpenGL.

Attributes are then cached upon first retrieval and will have almost no performance impact when finding an attribute many times.

***Note – Attributes are variables declared inside the vertex shader that are defined as 'in' variables, for example: in vec3 vertexPosition**

Example:

```
GLuint vertexAttribID = myShader->GetAttribute("vertex");
```

Getting Uniforms

All the **active** uniforms that the shader defines are loaded and cached automatically on program start.

Active uniforms are uniforms that actually affect the final color of the fragment. If a uniform is found to not benefit the program, it may be optimized away by the graphics card, and all attempts to retrieve it in C++ will fail.

To get a uniform, use this method:

```
GetUniform("Uniform Name")
```

This will return an unsigned GL integer (GLuint), and you can use this with raw OpenGL.

Example:

```
GLuint radiusUniformID = myShader->GetUniform("radius");
```

Light Buffer Block Index

If for some reason you find it necessary to modify the lights shader buffer, you can retrieve the index of the buffer using the following method:

```
GetLightBufferBlockIndex()
```

This will return an unsigned GL integer (GLuint) that points at the shader buffer storage that contains all the light data in the scene for the lit shaders to use.

This method is meant to be used by the light manager and should not be used in any other place, or the data may end up wrong.

Example:

```
GLuint lightBufferIndex = myShader->GetLightBufferBlockIndex();
```

Shader Properties

All the methods explained above that retrieve a variable from the shader are meant to be used with raw OpenGL. What if you don't need that? That's what **Shader Properties** are for.

Shader properties are structs that define a specific type of value that can be applied to a shader uniform.

To use a shader property, first create a shader property using its constructor:

ShaderProperty myProperty = { “PropertyName”, propertyValue };

The value type will be interpreted based on the value you pass to the constructor. If the propertyValue was a float, the shader property will define a float.

The supported property types are:

- Integer (int)
- Float (float)
- Matrix4x4 (glm::mat4)
- Vector4 (glm::vec4)
- Texture (Texture*)

After creating a shader property, to set it to the shader, first **bind** the shader and then use the following method to apply the property:

SetProperty(const ShaderProperty& shaderProperty)

The const ShaderProperty& part means that the shader property you pass in will be passed in by reference, meaning no copying is made, and it will be a const reference, so the method will not modify your property in any way.

Shader Properties for a shader must be reset every frame. Once the shader is unbound, the properties are lost.

Example:

```
Shader* myShader = Shader::Find(“lit/lit”);

ShaderProperty diffuseColor = {

        “diffuseColor”,

        glm::vec4(0.5f, 0.0f, 0.0f, 1.0f)

};

myShader->Bind();

myShader->SetProperty(diffuseColor);

… Draw Stuff …

myShader->Unbind();
```

Shader Variables

lit/lit

This is the standard shader that defines most of the properties that you are familiar with from Unity. This will be the most used shader, but when something doesn’t really need it, it should be avoided for performance reasons.

When using this shader with a material, or in code, the shader defines the following uniform variables -

Uniforms that are set through the Light Manager. These uniforms will have no effect when set from anywhere else:

Name	Type	Description
globalAmbient	Vector4	This is the global ambient light
fogColor	Vector4	This is the color of the fog in the scene
fogStartDistance	Decimal number	This is the distance at which the fog in the scene starts appearing from
fogDensity	Decimal number	This is the density of the fog

Uniforms that are set through a material or in code:

Name	Type	Description
materialAmbient	Vector4	This is the ambient color of the material. This defines the ambient color the material shows when light shines on it
materialDiffuse	Vector4	This is the diffuse color of the material. This defines the color that the material shows when light shines on it
materialSpecular	Vector4	This is the specular color of the material. This defines the color with which the material shines
materialEmission	Vector4	This is the color the material always emits. When there are no lights in the scene, this is the minimum color of the material
materialShininess	Decimal number	This determines how shiny the material is and affects the specularity
diffuseTexture	Texture	This is the texture of the model
normalMap	Texture	This defines the normals of the model
specularMap	Texture	This defines at which parts of the model it is shiny, and at which it's not

Materials

The Material File Type

Every material is defined in a **.mat** file.

To create a material file, create a new text document and change its file extension to

.mat

The first line of the file must define which shader you are using. This is done with the shader command:

shader shaderName

Once the shader is defined, the next lines define the values of the variables (uniforms) in the shader. These can come in any order and do not have to be in the same order that the uniforms were defined in the shader. This is done with the following syntax:

Name Type Value

- Name – The name of the uniform variable as defined in the shader
- Type – The type of the uniform. Supported types are listed below
- Value – The value to set the uniform variable to. The value must be of the provided type, meaning, if the type is a float, it has to be 1 number, if it is a color, it has to be 4 numbers

Supported Types

The supported variable types are case sensitive and must be one of the following:

- integer or int
- float
- matrix or mat
- texture
- vector
- color

When a type required multiple numbers as values, the numbers are separated by spaces.

An integer is a single whole number. Examples – 1, 17, 25, 0, -149
A float is a single decimal number. Examples – 1.0, 5.2, -49.3
A matrix is a 4x4 matrix consisting of 16 numbers. Example:
1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
A texture is the full path starting from the assets folder to the texture file. At the moment this only supports single textures. Cube maps may be added in the future. Example: mge/textures/bricks.png
Note – There are no quotation marks
A vector is an array of 4 numbers. Example – 1 0 5 1.3
A color is exactly like the vector, except that every number has to be between 0 and 1. Example – 0.5 0.2 0 1

Comments

When defining bigger materials, it may sometimes be useful to let yourself know what a certain value is or where it comes from. To do that, you can write comments. Comments are in human language and are ignored by the engine.

To create a comment, simply place a # in front of a line, and that line will be ignored.

Example:

#This is a comment

This is not a comment, and will be an error!

Example Material

```
#This is an example material defined in a file called exampleMat.mat

#First line must be a shader command!

shader lit/lit


#Skipping lines is fine

#Our material properties

materialAmbient color 1 1 1 1

materialDiffuse color 1 1 1 1

materialSpecular color 1 1 1 1

materialEmission color 0.1 0 0 1

materialShininess float 10


#Our textures

diffuseTexture texture mge/textures/BOOKS_TEXTURE.bmp

specularMap texture mge/textures/BOOKS_SPECULAR.bmp
```

Light Manager

The light manager defines certain properties that apply to the global lighting of the scene.

Config

To adjust the the configuration of the global lighting of the scene, you can change the values inside the **lightManager.cfg** file that is located in **"assets/mge/config/"**

The possible values are:

- globalAmbient – This defines the global ambient color of the scene. This is the minimal amount of light that will always be present, even when there are no lights. This is a Vector4, and should be **defined with 4 space separated numbers.**

- fogColor – This defines the color of the fog in the scene. This is a Vector4, and should be **defined with 4 space separated numbers.**
- fogDensity – This defines the density of the fog. The fog is calculated exponentially, and a good value is between 0.0 and 0.1. This is a float, and should be **defined with a single number.**
- fogStartDistance – This defines the distance at which the fog starts appearing. Anything before this distance will not contain any fog. This is a float, and should be **defined with a single number.**

Example Config

The light manager config file should look something like the following:

```
globalAmbient 0.2 0.2 0.15 1

fogColor 0.4 0.9 0.7 1

fogDensity 0.03

fogStartDistance 0
```

Level Editor

Building and exporting the levels will be done through Unity of any version above 5.4.

Installing the Editor

The first step is to install the editor package which contains all the components that the MGE supports and the exporter scripts.

1. Create a new Unity project and call it whatever you like
2. Double click the unity package called **MGELevelExporter.unitypackage**
3. Once the package is imported, a new option **MGE** will appear in Unity's toolbar

Creating a Level

To create the level simply place objects as you would normally when making a Unity scene. All the objects that you want exported should have an **ExportableGameObject** script on them. You can optionally add Exportable Components, like the **ExportableLight or ExportableAudio** to tell MGE that it should add these components to these game objects.

Exporting the Level

When you have a level ready for export, choose **MGE** from the toolbar and hit **Export Level.** Unity will then prompt you to save the scene if you haven't and then will ask you where you want to export the level creation script.

Here you should choose any folder inside of **"assets/mge/scripts/levels"**

Once done, MGE will be able to load your level.