Quick Shader 0.1 Beta

Documentation

(last update 2014-07-10)

QuickShader is a program that allows you to write and test shaders without creating your own rendering engine. Using this tool you can quickly create multiple graphic effects (e.g. phong lighting/shading, normal mapping, parallax occlusion mapping, pn-triangles and many more) that are used in modern games and simulations. Unfortunately program is restricted to shaders only, single pass pipelined effects. Effects based on postprocessing, blending, cubemapping and other techniques are currently unavailable (are planned for future releases).

Table of content

1. Warning notes 3
2. Application control.....3
3. Creating new effects 4
4. Parameterizing effects 4
5. Objects properties.....5
6. Build-in shader variables 6

1. Warning notes

Please be advised that QuickShader is a tool designed for shader programming. Writing unsafe code, like infinite loops, will most likely lead to the crash of an application or, in some cases, crash of the operating system. User is responsible for using this tool wisely.

Switching between effects on 3D models, without delivering appropriate data to shaders (for example textures) may lead to unexpected behavior (like application/system crash). For example there might be an effect that implements loop based on height map in one of the shaders. If the model using such an effect won't have a suitable texture assigned to the right texture unit, loop may turn into infinite.

Please note that OpenGL and GLSL don't have single uniform implementation. It differ between vendors. In some situations shader code that is compiling and running on one machine, may not compile and run on the other. Be sure to write code according to the GLSL specification.

2. Application control

LMB, MMB, RMB – left, middle, right mouse button click.

PLACE	KEYS COMBINATION	ACTION
Shaders (V,TC,TE,G,F)	RMB	Remove link to the file (shader from pipeline)
Scene	LMB	Camera look
Scene	arrows or W,S,A,D	Camera movement
Scene	Mouse scroll	Change camera movement speed
Scene	MMB	Selection
Scene	RMB + Y (by Y axis) + CTRL (faster) + SHIFT (slower)	Moving object
Scene	R + (X,Y,Z) + CTRL (different direction)	Rotating object
Scene	DEL	Remove object from the scene
Scene	L (increase) + CTRL (decrease)	Change size of the points
Global	F2/F3	Hide/Show interface
Global	CTRL + Mouse scroll	Change GUI size

3. Creating new effects

To create new effect click the **New Effect** button. New set of shader stages will be created in a form of a block shown in figure 1. Each shader in such a block can be created and programmed directly through built-in text editor. Shader stages currently in use are highlighted in red. To remove unwanted shader stage simply right-click on it. Once all desired shader programs are ready to be compiled and used in a rendering process you can click the **COMPILE** button. If there are no error messages in Log window, created effect can be used to render any 3D object on the scene, considering that it meets shaders requirements e.g. object must have two textures applied.

Figure 1. Single effect (block of shaders)

4. Parameterizing effects

To each effect custom global variables (uniforms) can be added. Those are really helpful in many ways as they allow the user to dynamically change algorithm behavior and its visualization simply by using the slider. To add custom uniform click **SHOW PROPERTIES** in the block you want it to be added and then click **ADD UNIFORM**. New uniform block will be created (figure 2) with an empty variable name field. This field represents name of the global variable. In order for shaders to be able to get corresponding value under that name, they will need to implement a declaration of it as follows:

For variable name “depth”: `uniform float depth;`

Of course compilation is required after such a source modification. Although no compilation is required if there was already such a declaration in code and user added corresponding uniform through interface. Application will find it at runtime.

Figure 2. Two uniform blocks with variable names: *depth* and *tessellation*.

5. Objects properties

Each object on scene has its own properties (besides points – those only have name displayed). Middle mouse button click on any 3D object will pop up a properties window as shown in figure 3.

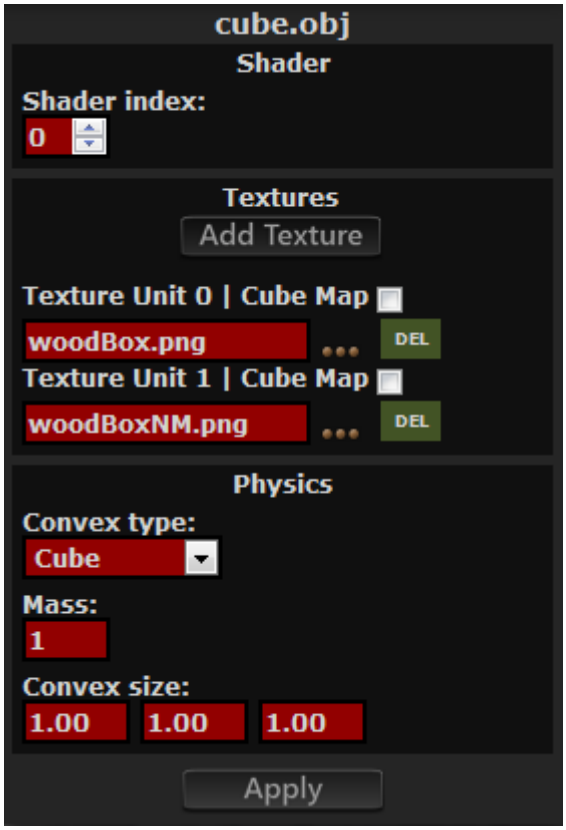

Figure 3. Properties of an object "cube.obj"

There are three sections:

Shader:
Shader index represents effect from the effects/shaders list. Each effect has its own index as shown in figure 1. User can change between effects by simply changing this value.

Textures:
User can dynamically add as many textures to the 3D object as he wants to, by pressing **Add Texture** button. Although there is a general maximum limitation, that correspond to the number of available texture units in user graphics card. It is also possible to assign a cube map to each of the texture units by clicking Cube Map check box. While cube map is set as enabled, multiple files can be selected in file browser. Each side of a cube map needs to have a texture assigned to it. Also to be properly created all textures that form it need to be the same size and format (RGBA/RGB). All texture files should be stored inside application local repository "Resources/Textures". For information on how to set texture units in shaders please see section 6.

(note: if application is crashing after adding image file, try to open that image with paint and save as a new file)

Physics:
Each object on scene has its own physics attributes for collision/interaction purposes.

- **Convex Type**:
  - **Cube** – object will be acting like a cube with specified size.
  - **Sphere** – object will be acting like a perfectly round sphere with specified diameter
  - **Mesh** – this type of shape will make collisions with such an object perfectly accurate (per triangle), but only when its mass is set to 0. When mass of an object is greater than 0 collision shape will be convex and treated as made of cloud of points/vertices (this may be very slow when object is made of large number of vertices).
  - **Plane** – it divides space in half thus creating impenetrable plane.
- **Mass:** Defines mass of an object. 0 means object will be static, greater than 0 dynamic. It is important for mesh convex type (see above). Each time the user starts to move an object, its mass is being reset to 0 in order to make it static and let user control the movement.
- **Convex size:** Available only for cube and sphere convex types.

(note: When exporting 3D object in modeling program it should be placed in the center of all three axes. Cube and sphere convex types are calculated from the local center of an object)

6. Build-in shader variables

Vertex attributes (vertex shader only):

```
layout(location=0) in vec3 position; // location 0 - position
layout(location=1) in vec3 normal;   // location 1 - normal vector
layout(location=2) in vec2 uv;       // location 2 - texture coordinates
layout(location=3) in vec3 tangent;  // location 3 - tangent vector
```

Matrices:

```
uniform mat4 MatrixModel;
uniform mat4 MatrixView;
uniform mat4 MatrixProjection;
```

Time span in milliseconds since application start:

```
uniform int Time;
```

Points (in view space):

```
uniform vec3 Points[n];              // n - number of lights on the scene
                                        (defined by user)
```

Textures:

- Option one:

```
uniform sampler2D TextureUnit0;
uniform sampler2D TextureUnit1;
uniform samplerCube TextureUnit2;
// etc.
```

- Option two (array of samplers):

```
uniform sampler2D TextureUnits[n];   // n - number of textures used by
                                                object (defined by user)
```

(**note from GLSL specification**: When aggregated into arrays within a shader, samplers can only be indexed with a dynamically uniform integral expression, otherwise results are undefined)