Useful features in FW

Kemppinen, Ollikainen
January 26, 2015

Introduction

The Nvidia framework (FW) is an efficient tool for multiple things: it handles standard linear algebra, file I/O for various formats offers a graphical user interface and API wrappers for CUDA and OpenGL. That said, it is rather intimidating; there is next to no documentation and not all of the features are immediately obvious.

Written as supplementary material for the courses ME-C3100 and ME-E4100, this document is not aimed at being a full documentation or tutorial on using the FW, but rather a description of some useful features that may be hard to find or figure out. The reader is expected to be somewhat familiar with the FW, but this is not strictly necessary since the described features can be understood on their own. They are also not necessary in implementing the assignment requirements.

Topics covered

1. Modifying the GUI
2. Texture I/O and usage with GL
3. Adding variables to state files
4. Adding missing OpenGL features
5. Adding support for new material parameters in the parser

1 Modifying the GUI

The GUI is built in the constructor of the `App` class using methods of the `m_commonCtrl`. There are three main elements: button, toggle and slider. All of these are similar in the sense that they are given a pointer to a variable which is modified when the element is used, keyboard shortcuts to the action and a verbal description of what it does. There is also a separator which can be placed to improve usability by grouping buttons.

Whenever a toggle or a button is pressed, the pointed value is set to the value of that toggle or button, given in the constructor. The value is either a boolean or an integer, where the integer is usually an enumerator. The only difference between a toggle and a button is visual: a button always looks the same, whereas a toggle indicates whether the variable is set to the value of that toggle.

A slider also modifies the pointed value whenever moved. They are given a range of values which can be either linear or exponential, determined by a boolean argument. The value can be either an integer or a floating point number. Sliders can also be stacked using the `beginSliderStack` and `endSliderStack` methods: all of the sliders created between those two will be in a single column instead of next to each other.

There are plenty of examples in the starter code of how these features work, and it is probably easiest to copy and paste those and play around a bit to understand how everything works.

2 Texture I/O and usage

To use images for any new purpose (such as for skydomes, projective lights or bokeh shapes), the `Texture` class is convenient. Most common formats are accepted – with the notable exception of `.jpg`.

For filtering it might be useful to fetch various MIP levels of a texture. They can be easily obtained via the `Texture::getMipLevel` method. It takes a single argument, the desired level. If the level already exists, it is returned – if not, it is generated and stored into a cache for later use automatically so you won't need to worry about storing anything. It doesn't matter on which level this method is called.

2.1 Import

An image file can be loaded as a `Texture` using `Texture::import`, a static method of the class. Note that it is easy to make the following mistake, producing unexpected behaviour:

```
Texture tex;
tex::import("cat.png");
```

This leaves `tex` uninitialized; import() creates an invisible local variable instead of storing the result in tex.

```
Texture tex;
tex = Texture::import("dog.png");
```

Here the loaded texture is correctly stored into `tex`.

2.2 Export

Export is also straightforward:

```
Texture tex;
Image img(Vec2i(100));
/* draw cat to tex and dog to img */
exportImage("cat.png", tex.getImage());
```

```
exportImage("dog.png", &img);
```

2.3 OpenGL

In order to use the loaded texture with OpenGL, the `Texture::getGLTexture` method should be called. It uploads the texture data onto the GPU if it is not there already and returns the `GLuint` name that OpenGL associates with the texture, used for `glBindTexture` etc.

3 Adding variables to state files

The state of the program can be saved and loaded using the FWs state file system. The system is modular and any class derived from `CommonControls::StateObject` can be added to the list written into the file. In practice, it's easiest to add any desired variables to the `readState` and `writeState` methods of the `App` class. At least the basic data types used with FW can be written and restored using the `StateDump::get` and `StateDump::set` methods.

4 Adding missing OpenGL features

The OpenGL binding of FW is relatively old and incomplete, so certain constants and functions will be missing. This is usually the reason if an OpenGL function call does not compile ('identifier not found') or some constant seems to be missing, even though the documentation states that it should exist. Adding these is easy.

For constants, copy the value from glext.h into `framework/base/DLLImports.hpp`, preferrably next to other similar `#defines` (though this is not strictly necessary; the value could even be used directly in the code where necessary).

For functions, find out the exact specification of the function in the documentation and then append a line in `framework/base/DLLImports.inl` in the following format:

```
FW_DLL_DECLARE_VOID(
void,
APIENTRY,
gl[Function name],
([parameter1type parameter1, parameter2type parameter2, ...]),
([parameter1, parameter2, ...])
)
```

The other OpenGL functions are at around line 400, but the important thing is that they're added inside any `#if !FW_USE_GLEW` block.

5 Adding support for new material parameters in the parser

If you implement a material model that requires additional parameters, first add them into the `Material` structure in `framework/3d/Mesh.hpp`. Then the values need to be added to the corresponding material entries in the `.mtl` file of the objects. In order to parse these additional entries into the `Material` structure, add a handler into the `loadMtl` function in the `framework/io/MeshWavefrontIO.cpp` file. The already existing parameters should provide sufficient examples of how to parse the new ones.