08/05/16
NCCA

COMPUTING FOR ANIMATION 2: GLSL IDE

Github Repository: 0Features-0BugsCVA3

Name	Github Account
Jonathan Lyddon-Towl	jlyddon
Adam Ting	yadang23
Jonathan Flynn	JFDesigner
Ellie Ansell	EllieAnsell
Philip Rouse	philrouse
Anand Hotwani	anandhotwani
Alexander La Tourelle	mainConfetti

ALGORITHMS & DATA STRUCTURES

Contents

1. Shader Manager Class

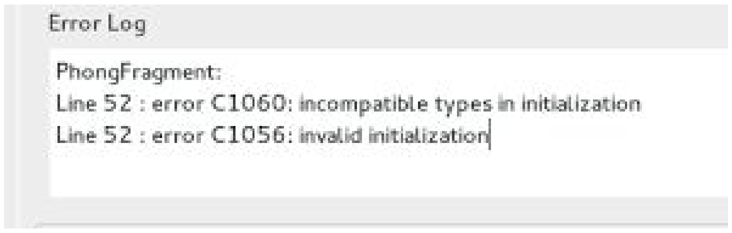
This class is created to utilise a modified version of NGL ShaderLib to manage creation and compilation of shaders to prevent cluttering the NGLScene class. CreateShaderProgram() is called when setting a new project. It creates a Shader Program object with the name of the project and attaches appropriately named vertex and fragment shader objects to it. The compileShader() function is used to recompile the current shader. The text from the QScintilla text editor are passed to it and the loadShaderFromString() function updates the loaded shader string. The shaders are then compiled and checked for errors. If they compiled correctly then they are attached to the shader program. The init() function is called when the program is first run to set up the initial phong shader and the default normals shader. The use() function takes as input an integer value to choose whether to use the current shader or the normals shader.

2. checkCompileError() and checkAllCompileError()

These functions check the code written by the user for any errors before updating the shaders in the OpenGL context. By blocking incorrect shader code from being updated, the user always will have a successful shader applied to their OpenGL context in the IDE to facilitate an iterative workflow.

Using a modified version of NGL, we have a function that can query the shader ID. Our version of NGL also prevents the program from exiting in the event the compilation fails. This means we can check the compile status of the shader itself before updating the OpenGL context. This allows the user to write their shaders and not worry about crashing the IDE if there are errors in their code. By using the “glGetShaderiv” function we can query the compile status. If the function returns “GL_FALSE” then the data structure needs to fetch information as to why the the shader has failed in compilation.

The error log is fetched and written to an std::string and then converted to a QString to be output in the IDE. The info-log prints out information on the error type and line number.



```
Error Log
PhongFragment:
Line 52 : error C1060: incompatible types in initialization
Line 52 : error C1056: invalid initialization
```

checkAllCompileError manages checking vertex and fragment shaders by placing them into a std::vector and iterating upon them.

When parsing the error Log, it was input as a QString. As a result the error had to be broken into individual lines. Each line was broken into a segments, broken up by “(“ and “)” , then the second element of each was the line number. The first element was replaced with “Line” to make the error log easier to read. Each line number was later used to highlight the line the error occurred, in the text editor.

3. Qscintilla Lexing

The QScintilla libraries include a base class from which to create custom lexers. The key function to reimplement from this is styleText(). This function is called every time text needs to be styled such as when a file is loaded or the text is edited. In order to correctly style the text the length, location and styletype of each word or symbol from the input is required, the characters themselves are not needed for the styling functions.
This work is offloaded to a Flex scanner in order to make the rules easy to read and modify. Flex also provides good optimisation and the generated scanner is fast. The only drawbacks were the initial learning curve to understand flex patterns and some additions to the .pro file to enable the flex scanner to build with the project and generate .cpp and .h files.
The styleText() function reads the data in the start to end range into a std::istringstream since flex require a stream input to lex. The lexer then loops calling yylex() and pushing back the style types, locations and word lengths into a packed array. SetStyling() is then called for each triplet of values.

4. Text Editor

QScintilla provides a lot of useful text editor features out of the box, but there were still extras added to make our editor faster and more helpful to users, such as block commenting with ctrl-/. QScintilla's SCN_CHARADDED signal is linked to a new slot to implement methods for automatically adding closing brackets and quotations as well as automatic indentations for {}.

Signals and slots are also used to allow the Error log parser to add error line markers into the relevant editor tabs.
The search bar is a QWidget created at the same time as the editor. The editor can then show and hide the search bar and signals from the buttons trigger the find functions. Scintilla's indicators are used to underline all occurrences of the search terms.

5. CebApplication

This class is used to control the GUI application's control flow and main settings. It inherits QApplication. The only additions I have made to this class is to allow for the GUI application to produce a message box if an error is thrown within the program. This allows for file errors to produce a message box when created. Whenever an error is thrown, the program halts and breaks from the function it was performing and shows an error messagebox.

6. CebErrors Class

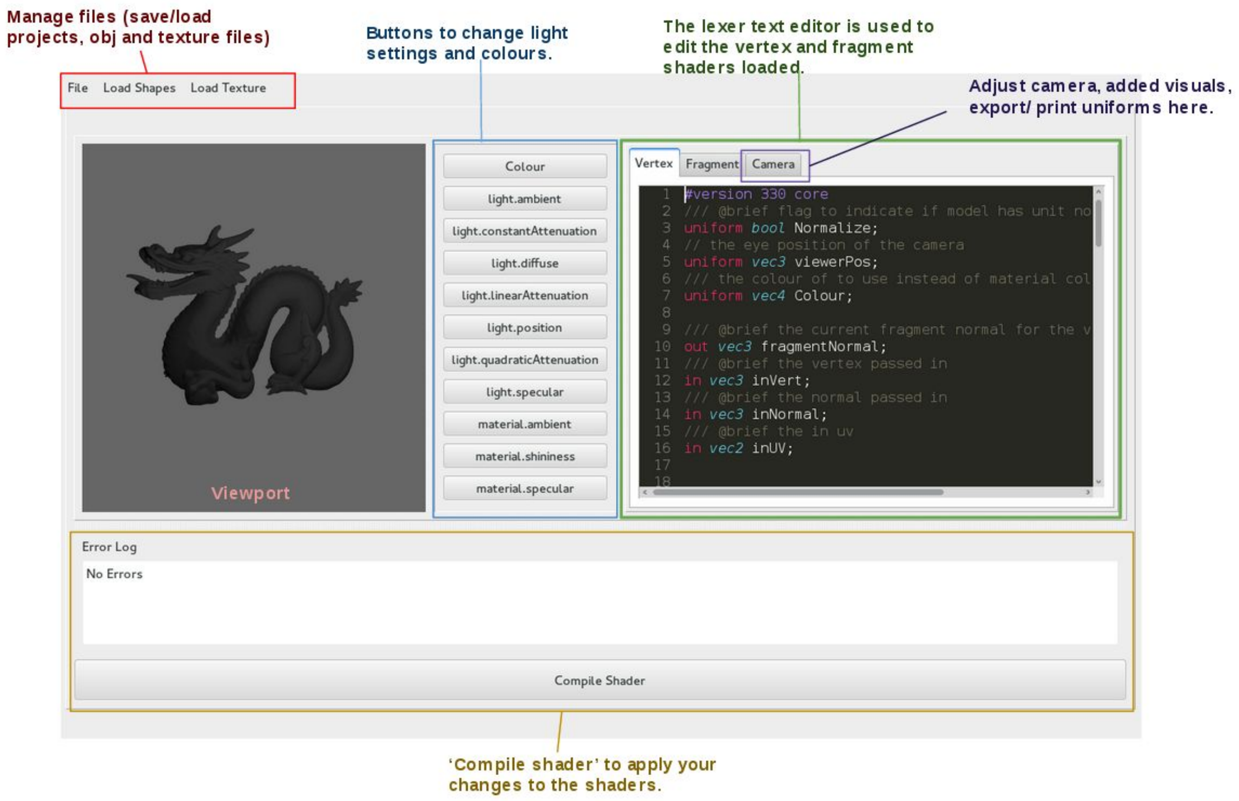
Contains a selection of different classes for throwing error exceptions. They all initially inherit std::exception and are fed into the GUI application control flow through CebApplication. The only real modification to each of the inherited classes are the what() function which explains what caused the error. In hindsight, I think it would have been better to have one base error class that took an enum instead of separating them out. I haven't had any experience doing error handling before but this was the best I could think of at the time. They do work nicely in tandem with the CebApplication and make it very easy to read when writing code since it's very clear what error you are throwing when writing it.

I also wrote a raise function for QtFileErrors so that whenever a file is opened it will display the correct error message. I begun to do this for OpenGL errors but found they weren't needed due to the decision of using an error log.

7. MainWindow Class

The main window GUI for the program. Most of the initializing work for the interface is done in the Qt Form Designer. The layout utilizes a horizontal and vertical splitter to contain collapsable segments of the GUI. The NGLScene, uniforms and Cebitor are all in separate sections as well as the Error Log at the bottom which allow a large amount of freedom in the UI design for the user.

A Tab layout is used to store the vertex and fragment shaders and a third tab used for controlling the camera settings.


Manage files (save/load projects, obj and texture files)
Buttons to change light settings and colours.
The lexer text editor is used to edit the vertex and fragment shaders loaded.
Adjust camera, added visuals, export/ print uniforms here.
File
Load Shapes
Load Texture
Colour
light.ambient
light.constantAttenuation
light.diffuse
light.linearAttenuation
light.position
light.quadraticAttenuation
light.specular
material.ambient
material.shininess
material.specular
Vertex
Fragment
Camera
Viewport
Error Log
Compile Shader
'Compile shader' to apply your changes to the shaders.

```
#version 330 core
/// @brief flag to indicate if model has unit no
uniform bool Normalize;
// the eye position of the camera
uniform vec3 viewerPos;
/// the colour of to use instead of material col
uniform vec4 Colour;

/// @brief the current fragment normal for the v
out vec3 fragmentNormal;
/// @brief the vertex passed in
in vec3 inVert;
/// @brief the normal passed in
in vec3 inNormal;
/// @brief the in uv
in vec2 inUV;
```

```
No Errors
```

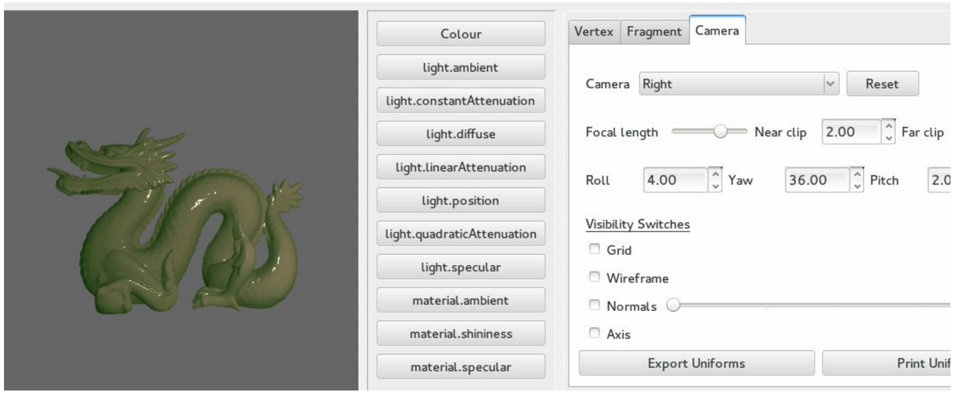
Fig. 7.1 Annotated main window

Custom GLSL files can be loaded in via the import vertex and fragment shader slots. Pressing the button in the GUI opens a QFileDialog where a file can be chosen for either the vertex or fragment. A QMessageBox confirms the replacement of the text currently in the text editors. If confirmed; the contents of the file are read into a QString via QFile and QTextStream and set to the corresponding text editor.

8. Camera

The ngl::camera was set in a separate class to NGLScene to avoid cluttering the class. As it's a QObject, signals and slots can be used to communicate between the UI, Camera, and NGLScene; when a widget's value is triggered such as FOV, a signal is sent from the UI, through a pointer in NGLScene to the Camera class. (The reason the slot wasn't connected directly to camera was because it was costly/ unnecessary to include camera.h into both mainWindow.h and NGLScene.h).
Its members such as createCamera() are called to create a vector of cameras ('perp', 'top', 'bottom' etc.) and the camera shape is set in accordance with the window's aspect, and other settings such as roll, yaw, pitch, near/far clipping can be set via the UI (fig.8.1). The active camera is assigned to an ngl camera variable in NGLScene which is used throughout, such as to find the MV position matrix. For interactive viewing, the Camera sends an updateSignal() to

NGLScene's update() function to update the paintGL scene so the user can adjust the settings to their liking with fast visual feedback.



Camera Right
Near clip 2.00
Roll 4.00
Yaw 36.00
Pitch 2.0

Fig. 8.1 Right camera view

9.Reading/ Writing files

Both Json and XML were taken into consideration for loading/ saving data, not only for the project file, but also to store sets of data (e.g. name, type, material). Its format was chosen for the following reasons:
1. The .xml format enables a customised set of tags, so it supports schemas and namespaces.
2. It's strict, so if the programmer wanted to access specific data stored in the file, such as the project name, they could query the name of the node where it would be found then access its value.
3. When writing, the node tree could be built in a structured manner by setting the name of a root node, then its children and their attributes. This limits the chances of errors as the data is uniquely set within it's own nodes.
Both are human-readable and easy to parse, but the main advantage was having the option to query data paths. Json is lightweight in comparison as there are less characters, and possibly arguably more readable when spaced correctly (auto Json is a good example of this when saving vertex and fragment data, see '10.Json') but wasn't expressive enough to separate attributes from values.

The project data is saved into an XML file format, which can be opened via the QDialog box and loaded later (see fig. 9.1, 9.2). The project name, vertex and fragment data are saved as a .xml in the user's directory and the data is accessible by accessing its DOM tree, and then is sent to project.cpp to be applied to the IDE. writeDataXML(_name, _type, _value) was also able to store data to be used elsewhere in the program.

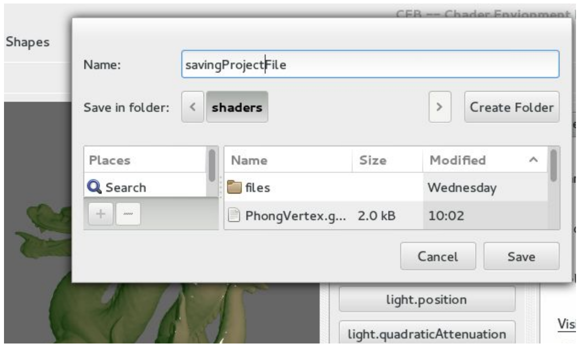


Fig. 9.1 Saving the project file

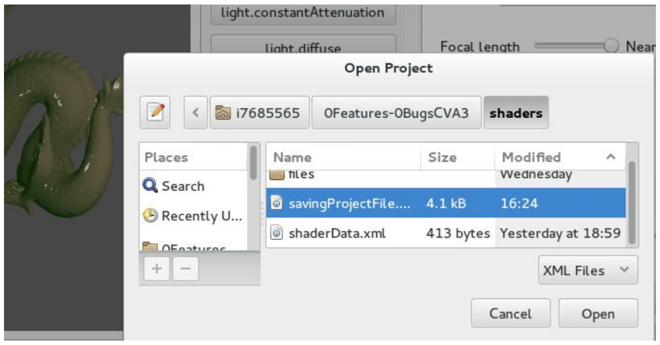


Fig. 9.2 Opening the project file

10. Json

Json was enabled to be used with the loadShaderFromJson() function by the shader project manager class however wasn't implemented in the final GLSL IDE. The Json file stores a string written in a format containing file paths to shader data, such as phongVertex.glsl and phongFragment.glsl. Each shader path sources the shaders which would be concatenated together to form a complete shader to load a string. These files could be contained with a directory (e.g. /shaders), as well as the version file. The default Json format file is initiated by defaultShader(), which uses Niels Lohmann's (https://github.com/nlohmann
) auto Json function to store it in a human-readable format. replaceWord(_oldWord, _newWord) can then iterate the string to replace the name of the shader, such as 'Phong' to 'Toon', and finds then loads the corresponding vertex and fragment shaders using loadShaderFromJson() in ShaderManager or NGLScene.

11. Project

Saving, loading and exporting projects is handled by the project class. It stores an instance of Io_Xml for obtaining and sending data to the project XML files as well as the project name and directory and a flag for the save state of the project this is initialised as false. The decision to store the vertex and fragment source directly in the XML files was made to ensure the data remained safe from external tampering (for example renaming glsl files. If the m_saved flag is false; the save() function uses a QFileDialog to set the project name and directory and then set the flag to true. The vertex source and fragment are passed in when the function is called in mainWindow. These are converted from QStrings to strings and written to an XML file with the internally stored project name using the internal stored project directory. The saveAs() function first sets m_saved to be false then calls the save() function. The load() function reads in the data from an xml file specified by a QFileDialog. It sets the project name and directory and outputs the vertex and fragment source strings by passing by reference. These are then set in the text editor from the loadProject function in mainWindow(). The exportProject() function takes as input a file directory from a QDialog and the vertex and fragment source strings from the text editor. Using QTextStream and QFile the strings are written to files with names being constructed from the project name in the specified directory. Confirmation of overwriting exported files is handled by a QMessageBox in the confirmOverwrite() function.

12. Parsing and Uniform classes

Parsing and storing the uniform data was done by finding the currently active uniform values, using glGetProgramInterfaceiv(...). This returns information about an active uniform. Each uniform is then stored in the class UniformData, in a std::Vector<UniformData>, where each element stores the name,location, type enum and string type name. Each UniformData has a child class for each respective value, e.g. UniformDataV4 is used for a UniformData that stores Vec4's. The functions of child classes have have been predeclared and are therefore able to be stored in the same std::vector<UniformData>. This allows the button classes to access the uniform values easily.

13. Button Library and Setting Uniform values

To create a variety of buttons for the uniform values, a parent class which held generic attributes and functions was created. Children classes such as ColourButton, FloatButton and VecButton then inherited the public functions and attributes from this class. The uniforms were passed out through Adam's parser, where the types and names were broken down and input into these buttons. Depending on the type of attribute, a different pop up window would appear. When these values were updated, the location numbers of the uniforms would be checked and as a result, attributes set. Upon compiling the shader again with new Uniforms, a duplicate of the current button list would be created, the button list would then be resized to 0 and the buttons would be created again. The values would then be copied across to these new buttons if the name and type was the same.

14. Button Classes

The structure of the buttons was made in a very straight forward manner. The Buttons were made as a parent and child relationship so that when storing the buttons in a list, they could all be of the same type within the vector. They had shared functions such as openBox(), which meant that this same function could be used each time, but with different outcomes upon being executed.

15. StartupDialog classes

The startup dialog is used to hold a few commonly used functions on the program startup. Recent Projects was not a feature that we could implement and so is redunent. The new project button takes you to the project wizard and open project allows you to start from a pre-existing project. A simple window, with it being almost entirely initialized as a Qt Form UI Design.

16. New Project Wizard classes

This file contains all the classes for the new project wizard. The new project wizard contains 5 pages, all with their own separate classes. The main wizard inherites from QWizard and the pages from QWizardPage. To make the process easier to manage and control the entire design was created from scratch and not from the Qt Form UI editor.

The project wizard works by adding each of the separate pages to the wizard and then registering the fields within those pages to communicate to the main wizard context.

The most noticeable section, code wise, is the use of the QTreeView widgets in the GlslFilesPage. These show the directory within the project */shaders/files/* and displays its contents in a single column. Selection models that were used to get the selected elements for use appending to create a new project. The only issue with this was that you couldn't get an ordered list from the selected items. This meant that we had to create a separate page for re-organizing the files in a listview so they could be appended into the file correctly.

Many aspects of this class design is untidy and would need to be re written through again to make it more efficient and less prone to break in the future. One main aspect is the use of the ItemSelectionModel's and FileSytemModels that we created in the Main Wizard and inherited into the different pages so that we could access the selection models between two pages. The pages aren't able to communicate with each other or to the main Wizard except through the register field methods. This meant workarounds were made to allow for this type of communication.