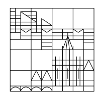
Modeling in Computer Graphics

Exercise Course

25 April 2018

Till Niese – `till.niese@uni.kn`



Setting up the Framework

To use and run the Framework following things required:

- And IDE of your choice that supports C++14
- CMake: a cross-platform tool to create project files
- GLFW: a multi-platform library for OpenGL
- A laptop that supports at least OpenGL 3.3
- Boost only required if the compiler does not support C++ filesystem
- Optionally libpng to read png images.

If you have problems with setting up the project then come to **Z1003 at the 02.05.2018 at 13:30**. Please send me an Email with the information what OS you use.

Popular IDEs with CMake support (selection).

IDE (Integrated Development Environment) that can be used in combination with CMake

	CLion	QtCreator	Visual Studio	XCode
Linux	x	x		
Windows		(x)	x	
Mac OS X	x	(x)		x
Free	30 days	x	x	x

Note: Using CLion does NOT work right now under windows, due to a problem with glfw. QtCreator under Windows and OS X requires the building tools to be installed. A free academic account for CLion can be requested using your `@uni-konstanz.de` email address at https://www.jetbrains.com/shop/eform/students

CMake

CMake is an **open-source**, **cross-platform** family of tools designed to **build**, **test** and **package** software.
CMake is used to control the software compilation process using simple **platform and compiler independent configuration files**, and generate **native makefiles and workspaces** that can be used in the compiler environment of your choice.
At least CMake 3.0 is required to build the project.

Installing the dependencies on Debian

To install all dependencies on Debian you can use **apt** with the command `apt-get install cmake cmake-gui glfw3 libpng-dev libboost-all-dev`.

You might also need to install the following dependencies `apt-get install mesa-common-dev libx11-dev libxmu-dev libxi-dev xorg-dev libxrandr-dev libXxf86vm-dev libglu1-mesa-dev`.

If your package manager does not provide glfw3 then you need to build it form source:

How to do this is described here

At the end you need to use `sudo make install` to copy the library to the right place

Installing the dependencies on macOS

To install all dependencies on macOS you can either use **macports** https://www.macports.org/ using the command

```
port install glfw libpng cmake boost
```

Or **homebrew** https://brew.sh/index_de.html

Installing the dependencies on Windows

Installing all dependencies on Windows:

- CMake: Download CMake from www.cmake.org
- Boost binaries can be downloaded here https://sourceforge.net/projects/boost/files/boost-binaries/1.64.0/. Depending on the IDE you use you need to choose the correct binary version. If you use VS 2015 then boost ist not required.
- The GLFW binaries can be downloaded on you should choose the correct bit version for your system. **The GLFW binaries only support VisualStudio up to 2015 right now. 2017 is not supported.**

Installing the dependencies on an other OS

If you OS is not listed here then write me an Email so that I can lookup how to install the dependencies for the system.

Build the Framework with CLion

Click on **Open Project...** in the start screen, or choose **Open...** in the menu and select the **CMakeLists.txt** file in the root of the Framework director. You should see a dropdown menu in the top right corner of the window, there you should select **simple** instead of **cgfw_core**

Build the with Xcode and Visual Studio

Open **cmake-gui** and use the directory where the framework is located (there is the **CMakeLists.txt**) as source folder:

- set the **where to build** to the subdirectory **build** of the framework dir
- run configure, choose compiler add paths, run configure, add paths
- generate
- open the generated project and build it using your IDE

Finished

If you program compiles and run then you should see a rotating coloured triangle.

If you have problems with setting up the project then come to **Z1003 at the 02.05.2018 at 13:30**. Please send me an Email with the information what OS you use.

Framework

The Framework provides on object oriented wrapper to the OpenGL API. And takes care about common pitfalls that might be encounter if the OpenGL API is used directly.

The naming of the classes and methods is as close as possible to the OpenGL API.

This should allow to easily adapt code that can be found on tutorial websites like:

- `http://www.opengl-tutorial.org/`
- `https://learnopengl.com/`

Pipeline Input and Output

The Input and Output of the Graphics Pipeline in OpenGL are:

- Vertices - Is the input unit to graphics pipeline (but might also be the Output in TransformFeedback or ComputeShading)
- Fragments - The fragments are written to the screen or an offscreen FrameBuffer and result in the pixel color.

Shader Program

A Shader Program is used to process the Input Vertices and creates the Fragments as output.

- `Vertex Shader` - receives Vertices as input and used to do transformations on the data.
- `Geometry Shader` (optional) - used to create additional Vertices out the data passed from the `Vertex Shader`
- `Fragment Shader` - creates the fragments/pixels that are output to the FrameBuffer/Window

For Transform Feedback the `Fragment Shader` is not used. The output of the shading pipeline is passed to output buffer.

Shader Program

The Shader Program is represented by the class `cgfw::gl::Program`.

The type of the shaders is determined by file extension.

```
auto programSolid = std::make_shared<cgfw::gl::Program>("Solid Shader");
programSolid->attachShaders("shaders/solid.frag", "shaders/solid.vert");
```

The Framework does hot reloading if a Shader was changed. So there is no need to restart the Program if changes are made to the shader.

Vertex Buffer Object

A `Vertex Buffer Object` is a memory container that holds the data that is passed to the Shader Program.

A VBO is represent in the library by the class `cgfw::gl::Buffer`. This class has a similar interface as a `std::vector`:

```
auto objData = std::make_shared<cgfw::gl::Buffer<VertexData>>();
```

Vertex Buffer Object

The data stored in a VBO can not be accessed the same time on the CPU side (Host) or on the GPU side (Device). To modify the data on the Host its usage hast to be *locked* for that: This can be either done explicitly using `std::lock_guard`:

```
std::lock_guard<cgfw::gl::Buffer<VertexData>> guard(*objData);
```

Or implicitly using the `update` method.

```
objData->update([](cgfw::gl::Buffer<VertexData>& data) {
  // update the buffer content
});
```

Vertex Array Object

A `Vertex Array Object` holds the information how the data of one or more `Vertex Buffer Object` is passed to the Shader Program.

```
struct VertexData {
    glm::vec3 position;
    glm::vec3 normal;
    // ...
};

auto objVao = std::make_shared<cgfw::gl::VertexArray>();

objVao->attribPointer("position", objData, &VertexData::position);
objVao->attribPointer("color", objData, &VertexData::color);
// ...
```

It represents the Object that will be rendered by the Program.

Vertex Array Object

The name passed as first argument to `attribPointer` corresponds to the name of the attribute in the vertex shader. The `&VertexData::position` is used to define which member of the struct has to be used for this attribute.

```
objVao->attribPointer("position", objData, &VertexData::position);
objVao->attribPointer("color", objData, &VertexData::color);
```

```
in vec3 position;
in vec3 color;
```

The name of the attribute and the member do not need to be equal.

Vertex Array Object

To render an Object the VAO has to bound, in the Framework this is done by passing the VAO to the `draw` method of the Program.

```
programSolid->draw(objVao, GL_TRIANGLES, 0, objData->size());
```

Uniforms and Uniform Buffer

Uniforms and Uniform Buffer are used to pass informations like View and Projection Matrix, LightPosition or any other global data to the Shader Program.

Uniforms can be set using the `set` method:

```
programSolid->set("model", model);
programSolid->set("tex", texture);
programSolid->set("shader_data", buffUniform);
```

Uniforms and Uniform Buffer

Uniform Buffers are like VBO, except that they only hold one element.

```
struct UniformShaderData {
    glm::vec3 cameraPos;
    float align1; // alignement value
    glm::vec3 lightPos;
    float align2; // alignement value
    glm::mat4 view;
    glm::mat4 proj;
};
```

The alignment the data in a Uniform Buffer is defined in the shader. To use the same buffer with all shaders `layout(shared)` has to be use. The data has to be aligned according the OpenGL standard.

Uniforms and Uniform Buffer

The Framework provides a functionality to check if the alignment of the fields of the uniform buffer is correct, and will report and error in the console if the alignment does not match.

```
buffUniform->uniformPointer("cameraPos", &UniformShaderData::cameraPos);
buffUniform->uniformPointer("lightPos", &UniformShaderData::lightPos);
buffUniform->uniformPointer("proj", &UniformShaderData::proj);
buffUniform->uniformPointer("view", &UniformShaderData::view);
```

Textures

The image data can be read using the `cgfw::utils::read_image_linear` function.

```
std::vector<unsigned char> imageData;
GLenum type;
unsigned int width, height;
GLint alignment;
glGetIntegerv(GL_PACK_ALIGNMENT, &alignment);

// read the image and linearize it
cgfw::utils::read_image_linear("image.png", imageData, type,
                               width, height, alignment);
```

Textures

This image data can then be used to create a new Texture and pass the data to it:

```
// create a new texture
auto texture = std::make_shared<cgfw::gl::Texture>(GL_TEXTURE_2D);
texture->setTexureWrapS(GL_REPEAT);
texture->setTexureWrapT(GL_REPEAT);
texture->setMinFilter(GL_LINEAR_MIPMAP_LINEAR);
texture->setMagFilter(GL_LINEAR);
// write the data to the texture
texture->image2D(0, GL_RGBA16, width, height, type, imageData);
```

Textures

To create a cube map Texture `GL_TEXTURE_2D` has to be replaced with `GL_TEXTURE_CUBE_MAP`, and the side for which the image has to be set has to be passed to `image2D`:

```
// create a new texture
auto texture = std::make_shared<cgfw::gl::Texture>(GL_TEXTURE_CUBE_MAP);

// ...

texture->image2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X,
   0, GL_RGBA16, width, height, type, imageData);
```

You have to set the images for all six sides otherwise the cube map texture is not complete.

Textures

After all images of the texture have been set the `Mipmap` has to be created.

```
auto texture = std::make_shared<cgfw::gl::Texture>(GL_TEXTURE_2D);
// ...
texture->image2D(0, GL_RGBA16, width, height, type, imageData);
// ...
texture->generateMipmap();
```

If the mipmap is not create and then the texture might appear black.

Textures

The texture then has to be passed to the shader program using:

```
rogramSolid->set("tex", texture)
```

In the shader, the texture is then made available as a uniform as follows:

```
uniform sampler2D tex; // a 2d texture
uniform samplerCube tex; // a cube map texture
```

GLM

The implementation of GLM is based on the OpenGL Shader specification. And provides basic data types like `glm::vec3`, `glm::vec4`, `glm::mat3`, `glm::mat4`, ...
That data types itself only hold the data. The manipulation of those data types is done using helper functions.
To get the length of a vector the function `glm::length` is used.

```
auto vec = glm::vec3(2,0,0);
float length = glm::length(vec);
```

The method `length` returns the number of components the data type has and is 3 for a `glm::vec3`.

```
auto vec = glm::vec3(2,0,0);
int numberOfComponent = vec.length();
```

GLM

The documentation of GLM can be found on `glm.g-truc.net`.

This site is currently not reachable from the University network.
We will upload the PDF to our website.

Exercise 1

- Become familiar with the framework and the glm library.
- Implement a camera that allows movement trough the scene using keyboard and mouse
- Implement a Skybox using a cube map

Due date **16.5.2018**.

Exercise 1

Listening to events in the framework is done registering event listeners to the window object:

- `MouseMoveEvent`
- `MouseButtonEvent`
- `KeyEvent`

```
wnd->addListener<cgfw::KeyEvent>([](cgfw::KeyEvent& evt) {
});
```

OR

```
void keyEvent(cgfw::KeyEvent& evt) {
};

wnd->addListener<cgfw::KeyEvent>(keyEvent);
```

Exercise 1

The `MouseMoveEvent` has four members:

```
double x;
double y;
double dx;
double dy;
```

`x` and `y` hold the current mouse location. `dx` and `dy` hold the delta of the mouse movement of the last mouse position.

Exercise 1

The `MouseButtonEvent` has this members:

```
double x;
double y;
// ...
int button;
bool leftButton;
bool rightButton;
bool middleButton;
EventAction action;
```

`x` and `y` hold the current mouse location. `action` has one of the values `PRESS` or `RELEASE`.

Exercise 1

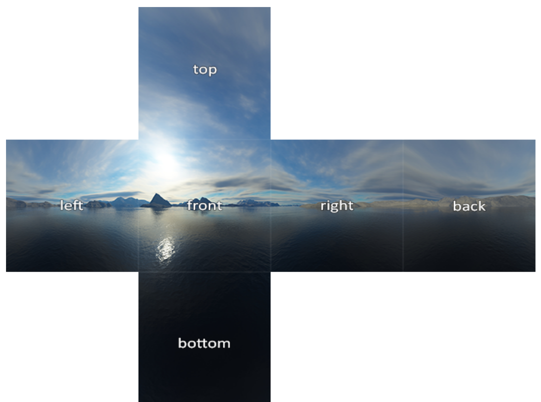
The `KeyEvent` has this members:

```
bool ctrlKey;
bool shiftKey;
bool altKey;
bool metaKey;
KeyCode code;
EventAction action;
```

`code` holds the information which ke was pressed. `action` has one of the values `PRESS`, `RELEASE` or `REPEAT`.

Exercise 1

A Skyboxes is an easy technique to create the illusion of environment in an infinite distance. It uses a six sided cube map, as a look up source:



Exercise 1

Creating a cub map is done using `GL_TEXTURE_CUBE_MAP` as texture target:

```
auto skyBox = std::make_shared<cgfw::gl::Texture>(GL_TEXTURE_CUBE_MAP);
```

And setting the data using `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_NEGATIVE_X`, ... to store the image data for the cube sides.

This can also be done using a for loop starting with `GL_TEXTURE_CUBE_MAP_POSITIVE_X`:

```
skyBox->image2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGBA16, width,
               height, type, imageData);
```

`https://learnopengl.com/#!Advanced-OpenGL/Cubemaps`