Contents

1 Topic Summary

1.1 Lecture Summary

- GLSL
- CMake basics
- GPU architecture basics
- steps to starting and ending an OpenGL program

- shaders and how to write one myself – `uniform` keyword
- vectors and matrices and how they are used to encode information
- transformations and Euler angles
- camera frustrum
- `y-x-z encoding` used in vectors
- how does the current code work and what steps does it take
- C++ memory management – pointer vs reference, smart pointers
- coordinate spaces and how they work together
- tree in `object.hpp`

1.2 Book Summary

2 Lecture 13.01.2020

- how to program graphics accelerators
- how we can use GPUs to render pictures on a screen
- real time graphics will be the focus
- we will attempt to build a 3D graphics engine (very simple of course)
- then we will try to write a simple game using this engine
- with each lab we will work to improve the engine
- we will use OpenGL ES to program our program
- what do common graphics engines look like
- how all the processing and communication works
- neither the development platform nor the type of GPU matters for this course
- we will work in C++ because it is the most common programming language for graphics programming
- it is incredibly complicated, but graphics development is generally relatively simple
- vectors and matrices and some C++ basics will be quickly covered in the first weeks
- GLSL will be used to program the GPU
- step by step improvements until we get to shading, colors, light, details, textures, texture mapping etc
- all other things are in the Syllabus

2.1 Basic Setup

1. update your graphics drivers
2. install `python 3`, `Visual Studio` (or another IDE), `cmake` – add `python3` to the path
3. install `conan` by running `pip3 install conan` in a shell
4. create a directory (named after your project, e.g. `lab01`) and navigate to it, create a directory called `build`
5. create `conanfile.txt` and add the following text:

```
[requires]
sdl2/2.0.10@bincrafters/stable
glew/2.1.0@bincrafters/stable
[generators]
cmake
```

6. create `CMakeLists.txt`, add this text (change *lab01* to your project name):

```
cmake_minimum_required(VERSION "2.8.0")
project("lab01")
add_definitions("-std=c++11")
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
add_executable(lab01 lab01.cpp)
target_link_libraries(lab01 ${CONAN_LIBS})
```

7. create `lab01.cpp` (must use same name as in `CMakeLists.txt`), add this code:

```
#include <GL/glew.h>
#include <SDL.h>
#include <SDL_opengl.h>
int main(int argc, char **argv) {
    static const int WINDOW_WIDTH = 500;
    static const int WINDOW_HEIGHT = 500;
    // SDL setup
    SDL_Init(SDL_INIT_VIDEO);
    SDL_Window *window = SDL_CreateWindow("lab01", SDL_WINDOWPOS_CENTERED,
        SDL_WINDOWPOS_CENTERED, WINDOW_WIDTH, WINDOW_HEIGHT, SDL_WINDOW_OPENGL);
    SDL_GLContext gl_context = SDL_GL_CreateContext(window);
    glewExperimental = GL_TRUE;
    glewInit();
    SDL_GL_SetSwapInterval(1);
    // SDL event handling
    for (;;) {
        SDL_Event event;
        while (SDL_PollEvent(&event)) {
            if (event.type == SDL_QUIT) { goto end; }
        }
    }
end:
    // SDL shutdown - opposite order of setup
    SDL_GL_DeleteContext(gl_context);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

8. run `conan remote add bincrafters "https://api.bintray.com/conan/bincrafters/publi` in a shell (source under 'Add Remote')

9. navigate to the `build` directory

10. *Mac OS:* run `conan install ..`
    Windows: run `conan install .. --build glew -s build_type=Debug`
11. *Mac OS:* run `cmake`
    Windows: use the CMake gui to set the `Where is the source` to your main project folder and `Where to build the binaries` to the `build` folder, click `Configure` and select `Visual Studio` (the version you have installed), click `Generate`, and finally `Open Project`
12. *Mac OS:* run `make`, then `./bin/lab01` to start the program
    Windows: in the sidebar of Visual Studio, navigate to your `.cpp` file in your directory, open it, click `Local Windows Debugger` in the top bar to execute

3 Lab 17.01.2020

- stuff that we need: python, cmake, conan
- some libraries:
  - SDL2 – we will use this one
  - GLFW
  - SFML
  - Allegro
- SDL2 is not natively supported in Windows and MacOS
- we need to find out what we can do with our drivers
- for this we will use GLEW or GLAD
- we will be allowed to copy and paste a lot of the boiler plate code in the exams
- go to the conan docs to find out what we need to do
- go to the SDL documentation for info on how stuff works
- **look up what conan needs for windows compilation**
- **set up all this stuff in CLion**
- **setup the whole path and compiler shit**

4 Lecture 20.01.2020

4.1 History of Computer Graphics

- tech was first used for military purposes *obviously*
- one of the first games was run on radar equipment to play tennis on a CRT screen
- Vannevar Bush (scientist at Los Alamos) postulated a system similar to our modern internet and inspired many other systems we use today
- *1963:* a scientist that was inspired by Bush was Ivan Sutherland who created a pen used for inputting stuff to a computer (“Light Pen” on “Computer Sketchpad”) – first CAD program
- *1968:* “The Mother of All Demos” called this because it showed windows, hypertext, computer graphics, GUI, video conferencing, computer mouse, word processing, collaboration in real time
- *1975:* “The Utah teapot” was the first 3D reference model used in computer graphics

- people got to thinking on how to simulate the real world, lighting, etc.
- *1973:* Phong shading algorithm was one of the first ones that kinda worked
- `Smalltalk` was designed at *Xerox Parc (Palo Alto Research Center)* to make developing GUIs easier and it was the first object oriented programming language
- *1984:* Mac enters the scene and has a lasting influence on computer graphics
- *1984:* Tron was the first movie that heavily used CGI
- *1985:* SGI develops a graphics API that eventually became OpenGL
- *1995:* Toy Story was one of the first entirely computer generated feature films, at that time is was still owned by Steve Jobs

4.2 Dedicated graphics chips

- the have a lot more cores
- the cores have less cache or share cache
- the whole architecture is extremely parallel, many pixels can be computed at the same time
- we have clusters of Streaming Multiprocessors which have clusters of cores with their own cache
- CPU designers: Intel, AMD, ARM, IBM, Qualcomm . . .
- most assembly for GPU is secretive because it is unique or might be a great advantage
- to interact with the card you have to go through a closed driver which does not reveal any proprietary information
- we have OpenGL, OpenGL ES, Vulkan, DirectX (Direct 3D libraries)

5 Lab 24.01.2020

- today we will discuss what is happening in the source we were given
- he showed how to use xcode with our development environment
- if you use an IDE we can also do debugging of the code which is really useful

5.1 Code Description `lab_02.cpp`

- initialize OpenGL
- create a window
- call glew to find out which functions our GPU supports
- set the refresh rate
- `vertex_shader_source`, `fragment_shader_source` are strings written in GLSL
- `vertex_shader_source` is used for geometry of the point and also for the color, *vertex shader* is meant to process this data. Here we don't do anything and just let it stay like it is
- `fragment_shader_source` is processing pixel data and it will generate the color of the individual pixels, this will be called for each pixel that makes up our triangle, it is interpolated
- the fragment shader is generally run in super parallel
- after that the shaders need to be compiled in order to run

- we link the shaders together, delete the now obsolete shaders and then get the location of the shaders in memory to be able to pass values to it
- we create an array of points that make up our triangle
- it is possible to pass most things to the GPU, we only need a few basic things
- then we will create all the buffers and pass the buffer data, giving it all the data and how often the data will change, here not very often
- we then call some functions that tell the CPU how to send the data to the GPU, like where is the position, where is the data, etc
- this means that we can use whatever layout that you want
- `stride` specifies how much data there is per point
- `position_attribute_location` starts at 0 because our points start right at the beginning
- `color_attribute_location` tells the GPU how far to jump to reach the first color values in a point
- `glClearColor`, `glViewport` tells OpenGl how to reset the screen in between frames and where to draw the 2D screen representation
- in the infinite loop:
  - `glClear` actually clears the screen, we just clear the color here
  - `glUseProgram` runs the shaders we programmed earlier
  - `glBindVertexArray` uses the specified vertex array
  - `glDrawArrays` finally draws the stuff, we specify the render type, how many vertices and when they start in the array
  - `SDL_GL_SwapWindow` *bufferswapping*, makes one buffer visible, the invisible one is then rendered to while the other frame is being displayed
- finally we do the cleanup just like in lab 1

5.2 Changing stuff

5.2.1 fragment shader – Line 46

- `gl_FragColor = vec4(1.0, 0, 0, 1.0);` to make it red
- `gl_FragColor = fragment_color + 0.3;` to everything brighter or whiter
- `gl_FragColor = fragment_color * 0.7;` to everything darker or increase the contrast

5.2.2 vertex shader – Line 37

- `gl_Position = position + vec4(0.5, 0.5, 0.0, 0.0);` moves triangle to top right
- `gl_Position = position + vec4(0.5, 0.5, 10.0, 0.0);` doesn't work because it leaves the drawing window and this is currently configured to not even have perspective

6 Lecture 27.01.2020

- vectors are very useful for computers and computer graphics
- in computer graphics 3D and 4D vectors are very common
- vectors encode direction and magnitude of something, anything really
- there are many representations: scalars generally italicized x and vectors generally in bold **a** or mathematically as $\vec{a}$
- vectors can also encode RGBA values
- geometric definition of a vector is an arrow pointing somewhere
- the zero vector does not have a magnitude nor a direction
- $-\vec{a} = \{-x, -y, -z\}$, multiply by scalar by multiplying all the elements by the same amount etc.
- for vector addition or subtraction, just add or subtract all the elements to/from their corresponding elements in the other vector
- we also have unit vectors and how to find the length of the vector
- we have the dot product for stuff
- then matrices are the holy grail of rotations and vector operations

7 Lab 31.01.2020

- matrices: n rows by m columns $n \times m$ matrix
- vectors: bold lower case **b** or $\vec{b}$
- matrices: bold upper case **M**
- matrix elements: matrix name lower case with subscripts m_{11} for matrix **M**, in programming we generally use 0, so it's m_{ij} with $i, j = 0$
- whether or not indices start from 0 or from 1 depends on the programming language or even between libraries
- identity matrices **I** are kinda like 1 in the scalar world, **I** multiplied by any matrix **M** yields **M**
- a vector is basically a row or column of a matrix
- row vector = $1 \times n$ matrix, column vector = $n \times 1$ matrix
- transposing a matrix = rows become columns – $m_{ij}^T \to m_{ji}$
- scalar multiplication: all components of the matrix multiplied by scalar
- matrix multiplication: $M \times N$ for `ncols(M) == nrows(N)`, resulting matrix will have `nrows(M)` and `ncols(N)`
- matrix multiplications are mostly used for transformations: translation, rotation, magnification
- matrix multiplication is not commutative, except for multiplications with identity matrices
- $\vec{a} \times B$ must conform to the same rules as $A \times B$
- matrices are very useful and generally pretty compact
- we can use $\vec{i}, \vec{j}, \vec{k}$ to get specific columns of a matrix, basically getting all x, y, z values, the generally have length 1 – **basis vectors**
- by modifying $\vec{i}, \vec{j}, \vec{k}$ we technically modify the whole coordinate system
- the rows of the matrix that contains the basis vectors are the basis vectors
- this can be use to scale and rotate objects – **transformations**

- we use Euler angles to specify angles – what are they?

8 Lecture 03.02.2020

- we'll step through all the code and learn what exactly it does
- how does glm represent angles?
- camera frustrum manipulation and all the values associated with it
- if near clip is too close floating point errors will happen – screw up the calculations
- adding small numbers to very large numbers can fuck up and either add to much or nothing at all because of the floating point standard
- we move stuff around in imaginary space, the camera has its own model space
- how can we arrive at the model, view, and projection matrices
- `uniform` in the shaders means we use the same mvp_matrix for all points
- we simulate all the stuff in 3D and then display all in 2D
- vertex shader is called after the vertex shader is done computing all its shit
- we retrieve the "address" of the mvp_matrix because we actually don't know where we should send the data
- y-x-z is a common rotation encoding, it is useful because it is often used

9 Lab 07.02.2020

- today we will work on lab_03
- find the bug and fix it: there should be two triangles on the screen, but there is only one
- we do we have so many windows and classes? it's inheritance
- bugs:
  - first bug in `es2_constant_material.hpp` – shader paths were incorrect
  - `glClear(GL_COLOR_BUFFER_BIT);` inside the loop is incorrect, calling it before the loop in `es2_renderer.hpp` draws both triangles **this was the problem**

10 Lecture 10.02.2020

10.1 Bug in new rendering code

- the main problem in the problem was `glClear(GL_COLOR_BUFFER_BIT)` being called in the loop
- another problem was having incorrect or missing memory management, `goto end` went to `end: return 0;` which is insufficient and will result in a segmentation fault

10.2 Memory Management in C++

- we have a stack and heap like in Java

- in c++ we can create objects where ever we want
- if we refer to stack variables outside of scope we will get a segmentation fault because there is nothing there
- if we put stuff on the heap, like the two triangles in our program, we need to free it before exiting out
- in Java the garbage collector takes care of unused objects (like triangles) and the memory will be de-allocated – not in c++ though, and gc is slow
- a gc stops the program and scans for stuff to free, but this is slow, in c++ you need to do that yourself
- the `triangles` all point to some of the same data, so removing that stuff would give an error and break the program
- `pointer` = points to some memory address, also `NULL`
- `reference` = also points to some memory address, but it must always point to something, not `NULL`
- `**var` is a pointer, `&var` is a reference
- references have a problem with object oriented inheritance: if we want to use inheritance in certain instances we cannot use references and must use pointers – problems might arise from that
- `c++11` added smart pointers which is a type of reference counting
- we can use `unique_ptr`, `shared_ptr`, `weak_ptr` classes that allow wrapping pointers and make them safer
- in the code `geometry` and `material` are raw pointers, when passed to `Mesh` we wrap them in `shared_ptr` in that constructor
- there we create two shared pointers for the triangles twice because of a coding error, when the pointer was removed for the first time all was well, but then the second copy of `shared_ptr` is decremented to 0 and the destructor tries to remove the stuff, and it crashes
- **see the recording or uploaded code for the new implementation of the shared pointers**
- on Friday we will talk about coordinate spaces – this will mean that we will need to restructure our code into an object tree that will help us use the program more efficiently – this will create pointers from parents to children and from children to parents

11 Lab 14.02.2020

11.1 Changes made to code

`shaders`

- added emission_color to have standard color
- adder point_size to make the point size modifiable

`aur.hpp`

- AU renderer
- umbrella header contains everything from a particular header
- common approach to make libraries more usable

`material`

- we can now set emission color and size
- now there is information about a dead shader that will the endless error messages

`objects`

- changes to getters and setters
- model matrix, world matrix: it is simpler to work with an object if it has its own coordinate space and then just put it together with the other spaces
- world space, object (or model) space, camera space: the common types of spaces
- looking through different lenses is simplified by having different spaces and putting camera and stuff into a common space
- any coordinate space can be encoded using a matrix
- we can put one object into a different coordinate space – useful for transformations related to other objects
- we can change model spaces and that is what parents are for in `object.hpp`
- the `requires_update` is optimization of matrix multiplications so that they are only carried out when it is necessary

`object.hpp`

- now is a tree
- root has `null` parent
- can have multiple children, share coordinate spaces
- to get the world matrix we step up the tree until one parent has one
- the tree needs to be traversed, right now the order is not considered or optimized, in reality we would do that
- now we consider the z-values of objects to figure out if some of them should be rendered or not

`camera`

- further optimizations

11.2 Testing

- adding rectangle as child of the triangle rotates the rectangle around the triangle as the whole world space of he world space is rotated

11.3 Exercise

- create a model of the solar system using the different coordinate spaces to make rotation easier
- circle vertices will be empty
- loop to 100 that adds vertices to circle by using sine and cosine to create a circle model
- keep the pi constant as float for safety
- `float angle = ...; x = cosf(...); ...; circle.push_back(Vertex{{x, y, 0.0f}}`

- factor out radius and vertex count vars for the circle
- radius of `0.5f`
- `auto circleGeometry = std::make_shared<ES2Geometry>(circleVertices);`
- `circleGeometry->set_type(GL_TRIANGLE_FAN)`
- `auto material = ::make_shared<const_mat>`
- `mat->set_emission_color(glm::vec4(...))`
- `auto name = ::make_shared<Mesh>(circlegeom, mat, glm:vec3(pos))`
- `name->set_name("name");`
- `name->set_scale(glm::vec3(x,y,z));`
- now we'll just make moon child of earth, earth child of sun
- they should all rotate accordingly now'
- it works and in real life a lot of things move in relation to each other

12 Lecture 17.02.2020

- geometry
- we used 2D shapes in three dimensions
- in OpenGL we have 3D space – how can we define 3D shapes?
- we can define these things by defining them the same way as 2D shapes
- we specify the triangles that make up the faces of the body
- right now we would need to specify shared vertices each and every time
- this is not efficient – we waste a lot of space and if we have more complex shapes it becomes super messy
- we can specify shapes by two arrays in OpenGL – one that just contains all the points and a different one that just defines vertices
- right now we use 7 floats for each vertex of our triangles – if we make a cube out of triangles there will be a lot of wasted space if we do not share the data
- we save all the indices with their corresponding coordinates
- then, we make lists of indices that correspond to faces of the bodies
- thus we reuse data and save space, especially if we use many vertices or if we store more than 7 values per vertex – 100 can be stored in certain engines
- `reinperpret_cast` is unsafe because we assume that vertex data is stored the same way as in an array – we can't necessarily know if that is the case
- `vertex_array_object` contains a description of what our data looks like because we could in theory be sending all kinds of stuff to the GPU
- in `es2_geometry.hpp` we send stuff to the GPU and tell it how the data is structured
- we now have a `vertex.hpp` class and that contains `glm::vec3 position`, `glm::vec4 color`, `glm::vec3 normal`, and `glm::vec4 texture_coordinates`
- normals are vectors that are perpendicular to the surface of our geometry
- normals are generally used to calculate light, to figure out which surfaces should be lit up or not
- in `geometry.hpp` we now have an unsigned int vector that stores the indices of the vertices
- we will just store all the indices one after another and then let the GPU figure out that three in a row make a point or whatever else
- `es2_geometry.hpp` now has more than just the buffer, vertices and stuff are stored

by the CPU while `GLuint index_buffer_object` are stored by the GPU

- we need to write some code in that class that only uses the indices and hopefully saves some memory – we need to update the `stride` and the other values – how many elements we need to skip to get to the next element, how to navigate the vertices
- most important call is `glDrawArrays` which actually draws all that stuff – now we use `glDrawElements` which uses the index approach instead of the vertex approach of `glDrawArrays`
- we will need small particles and circles in the future and thus `geometry_generators` was moved to a different namespace so that we can reuse it when we need to
- we have `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- when we generate spheres for example we can string together triangles in rings to make a sphere **see recording of the lecture**
- desktop GPUs support `GL_QUAD_STRIP` which strips together polygons of 4 vertices – it is supported by desktop GPUs but not by mobile GPUs – it is not often used because of that and it also comes with performance penaties