Contents

1 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

1.1 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

2 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**

3 Lecture 20.01.2020

3.1 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

3.2 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)

4 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

4.1 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

4.2 Changing stuff

4.2.1 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

4.2.2 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

5 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

6 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 – $m_{ij}^T \rightarrow m_{ji}$
- 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?

7 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

8 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**

9 Lecture 10.02.2020

9.1 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

9.2 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

10 Lab 14.02.2020

10.1 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

10.2 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

10.3 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

11 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

12 Lab 21.02.2020

-

13 Lab 28.02.2020

- in windows the driver works differently from macos
- in windows unused shader variables will be optimized and removed – causes some problems when some information is sent but there is nothing to receive them
- we can change resolutions to make it run faster or slower

14 Lecture 02.03.2020

- we talked about the midterms, for that see notes in Google Keep
- now we are talking about shading in a graphics engine
- we're talking about light and lighting sources
- now we have a new directory called lights that contains different light sources that we can use
- for directional light we only care about the angle of the light and not actually about the position of the light source because we are simulating the sun
- next is the point light which has some point of position
- explore this stuff and prepare for the midterm

15 Lab 06.03.2020

15.1 Midterm Tests

- target link libraries must be updated

15.1.1 Window Test

- just a blank window that shows that it works

15.1.2 Triangle Test

- hello world triangle
- try to animate the triangle, just move it
- and then change the colors of it

15.1.3 Geometries Test

- generate a bunch of stuff using the geometry generators
- just put them on the screen and animate them
- different color and other parameters
- different mesh types etc – just all the different options
- 3 planes, 3 circles, 3 spheres

15.1.4 Scene Graph Test

- just add stuff to parents and make a clock like in lab 5
- test the scene graph and modify it as you need to
- update the clock program for lab 6 because it breaks stuff

15.1.5 Lighting test

- have three spheres
- two of them are small and rotate around the other big one
- there might be a plane underneath the spheres that the light hits
- try to add as many lights as you can
- make the lights have different colors
- if it's a really cool scene we could get some extra points

16 Lecture 30.03.2020

16.1 Organization

- log in to Zoom was successful - I joined the meeting
- remote classes will most likely last until the end of the semester
- *we will not be able to conduct the theoretical midterm because we do not have the appropriate proctoring technology*
- **this will mean we will cancel the theoretical midterm and final – they will be replaced by practical coding tasks**
- no theoretical exams – just practical exams
- there will still be YouTube recordings uploaded
- Friday labs will be a lab at 10:50 because now there are no seat number limitations
- there will be no attendance checks though, so we can just re-watch the session

16.2 Lab 07

- no significant changes except for code cleanup
- code moved to C++ 17 because there are some nice features in there
- new directory called images: one checkerboard pattern, one just a photo (both creative commons)
- in lab07 we will add support for textures (images that shall be wrapped around our 3D objects
- other changes are minor cleanups
- **we added sdl2_image to the program** – maybe it's required to download that for me + add it to the cmake file
- we're now writing a texture_test program
- properly setting up working directories is important because we are now using shaders and images that we need to load

16.2.1 Setup Steps

1. create a plane to see that it is working

2. now we can write `auto[plane_indices, plane_vertices] = geometry_generators` to make is shorter → **update the helper functions**

3. create the plane using the aforementioned data
4. use ES2ConstantMaterial – we don't care about lighting
5. set camera position to fitting coordinates
6. run the program to see if it works
7. add a helper function to file_utilities that can read the pixel data of this image
8. create a `typedef` tuple of a vector of `uint_8` of width, height, number of channels; called `image_data_type`
9. create `imgage_data_type read_imgage_file(const string &path)`
   - check the documentation, `SDL_Surface *image`

     ```
     image = IMG_Load(path.c_str());
     #include SDL_image
     exit(-1); //plus error message if there is no such path
     ```

   - include the path in the message `auto bytes_per_pixel = surface->format->BytesPerPixel` if bytes count is not 3 or 4 complain and exit
   - now we return the data by first converting the data into a vector

     ```
     uint8_t *data = static_cast<uint8_t *>(surface->pixels);
     vector<uint_8> image_data(data, data + surface->h * surface->pitch);
     SDL_FreeSurface(image);
     return std::make_tuple(image_data, widht, height, bytes_per_pixel);
     ```

   - previously define `unsigned int width = surface->w, height = surface->h`
10. now we can create a texture

    ```
    auto[image_data, image_width, image_height, image_channels] =
    file_utilities::read(path);
    auto photo_texture = std::make_shared<ES2Texture>(image_data, image_width,
    image_height, imgage_channels);
    material->set_texture(photo_texture);
    ```

11. run the program and encounter an error – fix it by checking the fragment shader where the color from the texture will just be added to the color of the background – replace the addition by multiplication
12. now it works but the image is flipped because the picture has its own coordinate space too – botton left $\{u : 0, v : 0\}$, top left $\{u : 0, v : 1\}$, top right $\{u : 1, v : 1\}$, bottom right $\{u : 1, v : 0\}$
13. to fix the flipped picture – we could just flip it when reading it in (back to front) or one can change the way the coordinates are interpreted by OpenGL which would be the more appropriate way to do it; This can be specified in the plane generation where the texture coordinates get added to the vector and flip `u` and `v` there – **we can do it in both ways and there is no good or bad way to do it**

16.3 Homework

- can he control our computers and other forms of interactivity
- flip the image around so that it is displayed correctly in the program
- everyone that is able to do it will get +0.5 points – so do it and send him an email because I will not be able to join the session (probably)

17 Lab 03.04.2020

17.1 Zoom testing

- we are trying to shared screens over Zoom
- sharing the screen worked decently well – sharing full screen on windows does not seem to work well
- just sharing a specific window is not super effective because when we launch an application window it is not in focus and then won't be streamed
- Windows might not be giving access to the full frame buffer and stuff for Zoom because it wants best performance – so this might not be a bad thing, just a little bit annoying

17.2 New Additions to the code

- *we now have a UI subsystem that can help us debug stuff and change parameters while the program is running*
- he is using *ImGui* library to provide this UI subsystem
- `immediate mode graphical user interface = imgui`
- GUI popularized OOP to some degree, but this one is not, it is procedural
- surprisingly easy to use
- uses *immediate mode*: you give instructions and it works immediately – say you want a button and you get a button
- styling is obviously limited because that is not the point
- tons of input controls for colors and all that stuff
- library renders itself to a 2D texture that you can then put where ever you want – just put it onto a plane – thus it's simple to implement into our current graphics engine
- all user input is redirected to the library to enable input to the library
- very procedural in nature – see the github page
- simple but immensely useful library
- installing it is generally super simple because the library is so popular
- integration can be simple because there are pre-made versions for most combinations of OS and API
- we use OpenGL3.cpp and the SDL2_file: `example_sdl_opengl3` and then integrate them according to the instructions – maybe just wait for his code
- we render the library on a plane that covers our whole screen, all the input will also be rerouted through `imgui` because we need to control stuff through that

- inside of our infinite loop we put the code that creates the `imgui`
- the integration code will be given to us but the creation of the actual ui will be our task

18 Lecture 06.04.2020

- more talk about textures
- we have been given sample code to work with
- create a new test that he'll hopefully explain at some point
- test the code that is now lab 7 and then look into the `ImGui` implementation
- *texel* – pixel in a texture map
- texture mapping is totally up to the programmer today while earlier there were only a few specific options
- texels have (u, v) coordinates for each vertex
- in OpenGL $(0, 0)$ is in the bottom left by default but we can change that however we want
- DirectX has $(0, 0)$ in the top left corner, so it really depends on the API and how you code it
- we can now use 2 sets of texture coordinates to use
- Wrap Modes:
  - Repeat: repeats the textures using modulus
  - Clamp to Edge: repeats the last value (1) for all values ≥ 1
  - Mirrored Repeat: axis is mirrored which gives nice continuity in the textures
- Matrix:
  - first row: u max value
  - second row: v max value
  - fourth row: offsetting for x and y
- the matrix may also be used to rotate the textures
- the texture modes are set in the texture class and then submitted in the GPU where the actual calculations will be done
- textures can obviously be updated so that you can move it etc
- then the addressing modes are set in `glTexParameteri(...)`
- generally the texture coordinates are called (u, v) but in OpenGL they are called (s, t)
- when we are super close to texels we can see each individual one
- to fix this we may enable a filter that can improve the look by interpolation
- magnification and minification filter or interpolation modes:
  - nearest
  - linear
  - mipmaps
- we also get to choose the coloring mode:
  - addition
  - multiplication
  - mix of two
  - subtraction
  - reverse subtraction

- now textures also work with phong shading so we can update all that stuff now
- multi-texturing: applying more than one texture to a surface at one time
- these textures can then be combined in different ways and enable many different looks

19 Lab 10.04.2020

- new test that will be on final exam
- we will have to modify the engine ourselves this time
- we can open pictures in CLion to look at them
- we want to minimize the amount of data that needs to be streamed from GPU memory to GPU processor because textures tend to be pretty large – some methods are:
  1. keep the texture compressed – generally hard to sample without decompressing so not used
  2. use a special format for the texture that allows sampling while the texture is still compressed, these are called block compression – uses all kinds of tricks and is generally not as good as JPG or other dedicated formats
  3. use mipmaps that are basically textures of smaller resolutions that are created on the GPU – sampling a smaller mipmap is more efficient and that is generally used for objects that are further away, based on distance from the camera – we can easily ask OpenGL to do this for us, mipmaps can also be loaded from disk
  4. another optimization is *anisotropic filtering* – if you scale the botton checkerboard you can see the bad edges. If you enable some filters it looks better but it's still not great. Anisotropic filtering takes into account the angle that the texture and camera have to each other to fix some of the artifacts
- anisotropic filtering: just one extra GL parameter that we have to supply
- textures are just 2D arrays that shaders have access to and then do with whatever they want
- normals are used for light reflections off of surfaces – in between vertices the values are interpolated to make up the data so that every pixel (texel) gets a normal
- if we just repeat the normals as a simple form of interpolation, we don't get a 3D textured look
- if the normals change to reflect the 3D texture of the object, we get tesselation or Bump Mapping
- modified lab where we have a point light circling above a plane
- now the normals are properly interpolated leading to Bump Mapping
- if we enable bumps we suddenly get a nice 3D looking brick surface that tanks performance but looks really cool
- even though we only have 4 actual vertices, it looks like we have thousands
- task: **add normal mapping and bump mapping to the engine for the final exam**
  - the book contains code snippets that we need to adapt
  - normal map texture generators are available online
  - there are also procedural texture generators that make it simple to get textures
- low poly objects together with good normal mapping can make stuff look almost

like high poly objects

20 Lecture 13.04.2020

- bump mapping or normal mapping was discussed last time
- one of the things we have to do before the final exam: add normal or bump mapping
- today we'll go through the chapters of the book and find out what we have to modify in our engine
- **read texture mapping (10.5) and then bump mapping (10.9)**
- real height mapping is really good, but it is super expensive, also called height mapping – the vertices themselves are what is moved and our surface itself gets real details
- normal mapping improves the look less than displacement mapping but is a lot faster
- there are many programs that will help you create normal maps of certain things
- `NormalMap-Online` is a nice, free, online tool to create normal maps
- the normals should be encoded in a way that allows the normals to be correct regardless of what orientation the texture is applied in
- *tangent space* is used for that: +z points away from the surface (same as normal), x basis vector: in direction of +u, y basis vector goes into +v
- *tangent space* will be encoded in a matrix that can then be used for transposition
- we need to calculate the tangent and binormal vectors on how to do that stuff
- the procedural generation is pretty easy for most objects, and for most other things this data is pre-generated to save time
- **look at Some HLSL Examples (10.11) for code examples that are pretty much transferable**
  - instead of HLSL `struct` in input we use `attribute` in GLSL
  - instead of HLSL `struct` in output we use `varying` in GLSL
  - `uniform` are values that are the same for all vertices in the system
- look through the shaders and see what is used for tangent space and transfer that one
- pixel shader == fragment shader
- implementation
  - put the file into `./include/geometries/`
  - add tangents and determinants into `vertex.hpp`, find the binormal in vertex directly
  - use `glm::vec4` for tangent: `x, y, z, det`
  - create `struct` triangle or sample vertices 3 at a time – find out which is more efficient
  - every 3 consecutive elements in `_indices` in `geometry.hpp` make up a triangle
  - making a class can allow real-time modification of goemetries
  - we need the function `computeBasisVectors()` in our `geometry.hpp` but rename it to `computeTangentVectors()` to make the purpose clearer
  - after finishing the tangent generating function we need to submit the data to the GPU
  - for this we need to modify `es2_geometry.hpp` in `gl_buffer_data`, we need

  to add 4 extra elements as tangent and put all the correct values there
  - add the texture – we need a separate type of texture for normal values
  - go to `es2_phong_material.hpp` or also `phong_material.hpp` or `texture_1_normals`
  - think about adding normals to texture_2
  - add the required code to the shader so it knows what the data means
  - modify the phong shaders accordingly to the examples in the book
  - whenever the light vector is used in our shader we use a similar transformation to the one we have in the book
  - to quickly check of normal mapping works: convert the normal vector instead of the light vectors
  - `tangent_binormal_normal` put into one matrix and multiplied with `view_normal`

21 Lab 17.04.2020

- new additions to the engine
- normal mapping code is for bump mapping and textured plane test is for pure textures
- new test as material properties test: colors, depth testing, culling, blending – two lights that circulate around viewers head, red sphere, checkerboard background, sphere also has checkerboard
- **take notes on the things that we talked about**
- play around with the *Imgui* thing that he provided
- most things are added to the materials class, a bunch of bools and enums that specify behavior
- stuff is not submitted in `material.hpp`, es2 is made for that
- one might be able to inherit parts of `es2_phong_material` from `es2_constant_material` if one wanted to,, but that sounds like a clusterfuck
- in the material all the bools are used for `glEnable(<feature>)` depending on what is specified
- all of these variables that we specify are generally stored in the GLContext pointer
- depth testing is important to figure out if objects are actually in view or if another thing is in front of it, the color will not be drawn. Sometimes objects should also be drawn even if they are obscured for some kind of super vision or something
- depth testing is only used to determine if color is supposed to be shown and thus for whether or not it is visible
- face culling is removable geometry that is not drawn because it is not visible
- this is done through winding order, the direction in which the vertices of a triangle are numbered in – clockwise or counterclockwise – and that differs between looking at the front or the back of the face. This allows the API to discard inner faces of shapes and make drawing easier
- if one wants to see inside of objects the culling face can be swapped from front to back to only render inside faces
- blending = managing the transparency of objects and stuff, especially if they intersect or are in front of each other
- the blending stuff is also available in our lab

22 Lecture 26.04.2020

- fog that gradually hides stuff in the distance
- global property of the scene generally, but we go with granular approach
- is now property of the material and then can be enables etc
- we have exponential, exponential squared, and linear fog
- linear fog is pretty simple, just linear increase in fog from distance x to distance y
- lamps currently don't have any fog acting on them
- nice an accurate control and cheap but not super realstic
- exponential is exponential, while the squared exponential is more like a linear curve with a bit of curvature at both ends
- there are some different ways to measure distance from the camera that are then fed into the fog details
- flat depth is not the most accurate thing because it does not take the precise distance from the camera but the distance to a z-plane
- radial depth calculation is better fitted but it is more expensive to calculate
- just play around and figure out what the things do
- another approach is to not apply fog to every pixel but to apply it to every vertex to speed up the calculations
- the last part will be optimizations
- modern engines attempt to simulate atmospheric scattering too but that is super complicated
- precomputed atmospheric scattering is also a thing that some people are doing
- we won't be asked to do anything with the fog
- **project:**
  - next week he will tell us exactly what we are going to be required to do
  - collision detection, rays, stuff
- the kind of games we will try to emulate is the original doom *The Ultimate Doom* – one room with walls around, add textures, some enemies, weapons, that kind of thing – maybe just a shooting gallery of monsters that go towards to player – we can do whatever we want, this gives a lot of extra points
- we will use doom because the geometry of it is very simple and that lends itself to our imitation – everything is done in 2D sprites
- enemies in doom always turn towards the player and look at them – we need to rotate the enemies accordingly

23 Lecture 01.05.2020

- just use some version of the engine to get bump mapping working
- there are sources in lab 08 that we can play around with
- move everything to lab 08 code level and build:
- bump mapping
- doom clone:
  - sprites are just 2D, there is not a lot of 3D going on
  - the planes always point towards the camera
  - *billboards* are planes that always rotate towards the camera

  - smoke and particle effects are generally planes
  - mouse control of camera is to be implemented – in `sdl_window`
  - of `mouse button event` and `mouse move event` – we get absolute x, y‘ and relative x, y from last frame – nice for moving screen
  - set mouse mode and capture mouse to `true` – `SDL_CaptureMouse` and `SDL_SetRelativeMouseMode` at `15:00 min`
  -
- Tests:
  - [ ] general usage test
    * [ ] use lights as slower moving bullets
    * [ ] ground plane with lamps and textures to set the athmosphere
    * [ ] copy camera move code from Lecture 10 `20:00 min`
      · move in the direction that the camera point in
    * [ ] window set_on_mouse_move for camera rotation at `22:00`
      · add sensitivity setting in globals
    * [ ] add billboards that move towards the player
      · look_at_matrix is a function `28:30` code added to es_2_material
      · need to disable linear filtering – otherwise is looks like mush
    * [ ] set movement to be on the plane always
    * [ ] make random spawn for monsters
    * [ ] simulate steps using `sin` and `cos`
    * [ ] create a full billboard like at `37:00`
    * [ ] figure out line-of-sight tracing
    * [ ] figure out collision tracing for shooting
  - [ ] lighting test
  - [ ] material properties
  - [ ] normal mapping test
  - [ ] other geometries test
  - [ ] scene graph test
  - [ ] textured plane test
  - [ ] triangle test
  - [ ] window test

24 Lab 01.05.2020

- notes on shot tracing and collision detection
- generate ray from screen through frame
- iterate through all elements and see which one is hit
- optimize checking by like naming object specifically and then deciding what should happen
- a ray class will be added that goes from some origin into a particular direction
- ray: line with origin point; could give it certain length and use it for projectile movement
- encode the direction in unit vector and then get a function that returns a point at a certain distance
- see how to at `18:00`

- collision algorithms that check all kinds of geometry are cool but expensive – won't use them
- we check with simplified geometric objects
- you encode an object by a sphere for example – algorithms are well know
- axis-aligned bounding boxes are one step up and algorithms are well known too
- good example for sphere intersection is `viclw.github.io`
- intersection test in ray – gives intersection and distance
- helper method in camera class that creates our ray
- write nice optimized code that makes sense and create helpers to create all these elements
- check whole video for examples of how this works
- at `1:05:00` we get a good view of a lot of working code

25 Lecture 08.05.2020

- Final Exam Discussion:
  - exam should be postponed by one week if he can
  - seniors will need to have their grades by the 18th, so they will have the exam on the 15th
  - seniors will have their exam on the 15th of May – they'll get an email
  - juniors are gonna have the exam on the 21st of 22nd of May – awesome
  - details will cone this week on the repository
  - exam will work in a similar manner as the normal exams
  - juniors on 22nd or 21st – normal mapping, final project and porting all files to the very final version
  - lots of useful stuff in the final version.
  - port all the files over, port all the helpers and create more
- how does one add a texture for a gun or something?
- we can parent stuff to the camera, but that might not have the desired effect
- create walls all around, add bump mapping to walls
- different sprites for shooting, lights
- pay attention to order of submission of sprites, use distance to camera
- the drawing is done with sorting by different characteristics
- mobile GPUs have been doing a lot of ray tracing for some time
- guns is rendered as an overlay, overlays are not rendered with `view_matrix`
- submit imgui as a texture to make a menu and overlays
- object now has: `look_at` – orients the object as something
- implement some multi-threading?
- billboard_toward_camera to set biilboards correctly

26 Lab 08.05.2020

- now shooting works and sprites are animated
- sprite sheets are used for optimizations
- we zoom and transform the texture using the transformation matrix

- this is used to put the right texture in the right spot
- text is difficult, generally libraries are used for that
- find some ways to display text
- use the enemy class
- texture frames are submitted to the enemy class
- how can one use sounds in our engine?
- shooting works through mouse down, unsurprising
- where can we get sprites from?
- use tons of nice textures and maybe write a struct that contains info for the sprite sheets
- minimize all the GL calls
- implement some kind of fps counter
- merge geometries together to improve performance
- look into instancing of common objects
- avoid if-statements as much as possible