

# CS-C3100 Computer Graphics, Fall 2024

Lehtinen / Härkönen, Kempainen, Kynkäänniemi, Kozlukov, Timonen

## Assignment 1: Introduction, Due September 22, 2024 at 23:59.

### 1 Overview and Requirements

This assignment will guide you to the coding environment we'll be using for the semester, and gently introduce you to drawing, generating, and moving 3D models.

**Requirements (maximum 10 p) on top of which you can do extra credit**

1. Moving an object (1 p)
2. Generating a simple cone mesh and normals (3 p)
3. Converting mesh data for OpenGL viewing (3 p)
4. Loading a large mesh from file (3 p)

### 2 Getting Started

- On Windows, make sure you have Visual Studio 2022 (the freely available Community Edition suffices) installed with C++ development options enabled ([link](#)).
- Fork the assignment repository at [Aalto GitLab](#) and clone it to a suitable location on your machine:
  1. Pick a good location to store your code in. If you are using a virtual Windows instance on `vdi.aalto.fi`, make sure you use a persistent storage like your home directory that is mapped onto the `Z:` drive. Otherwise you risk losing any work when the ephemeral storage on the instance gets destroyed upon shutdown. (Not that reasonable students would stop their working sessions without committing their code to remote source control, anyway.)
  2. After signing in with your Aalto credentials at the above URL, click "Fork" in the upper right corner.
  3. In the screen that comes up, make sure the fork gets placed under your own username and not any GitLab group you might be member of.
  4. *In the same screen, make sure you set your fork's visibility to "Private". This is important.*
  5. Navigate to your shiny new fork and clone the repository. On Windows, this happens by opening "x64 Native Tools Command Prompt for VS2022" from Start Menu, navigating to your location of choice, and running `git clone` with your repo's address. On a VDI virtual machine, this could look like

```
C:\Program Files\Microsoft Visual Studio\2022\Community>z:
Z:\>cd studies
Z:\studies>git clone https://version.aalto.fi/gitlab/[USERNAME]/cs-c3100-2024.git
```

6. We strongly recommend setting up SSH public key authentication to use with GitLab so that you do not have to enter usernames and passwords every time you pull or push code.
- Open README.txt, inspect the contents and fill in what you can at this point. You will turn in this file together with your code. Update it while you work on the assignment, like a to-do list.
  - Optionally: go to the MyCourses system and make sure you can submit files to your "Assignment 1" folder. You can submit as many times as you want; we will inspect the last submission before deadline.
  - Try out the provided reference.exe executable. Your program will work like this when it fulfills the requirements. The example also implements a few minor extra credit features. *Note that it only runs on the Windows platform, so if you're working on another platform, you can try it out in a virtual machine at vdi.aalto.fi.*
  - Prepare the code for building using the CMake build system generator.

On Windows, the process is as follows:

1. Open "x64 Native Tools Command Prompt for VS2022" from Start Menu.
2. In the command prompt, navigate to the directory assignment1 cloned from GitLab and run the CMake configuration step for the configure-vs2022 preset. If, for instance, you cloned the repo on a VM into your home directory under studies\cs3100\assignment1, this could look like

```
C:\Program Files\Microsoft Visual Studio\2022\Community>z:
Z:\>cd studies\cs-c3100-2024\assignment1
Z:\studies\cs-c3100-2024\assignment1>cmake -S . -B build --preset=configure-vs2022
```

The last step downloads the necessary external libraries using the vcpkg package manager, and generates a Visual Studio solution file build\assignment1.sln.

The vcpkg configuration that we provide also includes presets for generating XCode projects on MacOS, as well as presets for building Debug and Release builds directly for use on, e.g., Linux systems. We do not officially support other platforms than Windows except in a best effort fashion. Like with all content-related questions, we recommend asking actively on the assignment's Slack channel for pointers and help if you run into issues.

- In Windows explorer, double click the file build\assignment1.sln. This should open the code in Visual Studio 2022. Choose 'Build → Build Solution'. Once the build finishes, run your program by pressing F5. You should see a triangle floating above a reference plane, like in the reference. Try clicking on the buttons in the control window to change the geometry and shading mode.
- At least skim the rest of this PDF before you start coding, especially Section 6 about submitting the assignment!

## 3 Tour of the environment and the code

### 3.1 Environment and tools

The Aalto classrooms should have Visual Studio 2022 installed. For working from home, you can get the free VS2022 Community for Windows Desktop ([link](#)). This requires signing in on a Microsoft account but is free.

The `assignment.sln` file you opened is a VS2022 *solution*, essentially a workspace. It contains three *build targets*, of which `assignment1` is the one relevant to you. You should never need to modify the project settings or add new files to complete the requirements, but you may do so. If you expand `assignment1` in the Solution Explorer, you see folders for source and header files. You should only need to modify `App.cpp` and `App.h` to complete the requirements.

There are four *build configurations* in the Visual Studio solution you can choose from: Debug, Release, MinSizeRel, and RelWithDebInfo. You don't need to care about the last two. Debug has more error checking enabled, so start with it. If your program feels slow, always try running it in Release before assuming there's something wrong with your code; Debug builds can sometimes run ten times slower than Release builds. While working with larger data, you'll probably want to use Release and switch to Debug only while tracking down problems. Sometimes your code has a serious problem that causes Debug builds to crash while Release builds *seem* to work; don't ignore these problems, fix them!

### 3.2 Libraries, frameworks, and standards

We avoid introducing technology for its own sake; you are supposed to be learning the theory via the practice. We try to make it so that whatever technology you encounter is standard and widely applicable, not just useful on this course.

We make heavy use of the C++ standard library (STL) – this allows us to concentrate on the important things instead of reinventing the wheel too often. You'll recognize standard library code because it resides in the C++ namespace `std`, for instance `std::vector`. In addition, we use the following cross-platform libraries through the `vcpkg` C++ package manager:

- `GLFW` for window creation and management, as well as handling keyboard and mouse input;
- `ImGui` for immediate-mode graphical user interfaces;
- `Eigen`, a header-only template library, for linear algebra;
- `fmt` for formatting strings in a modern fashion<sup>1</sup>;
- `nativefiledialog-extended` for portable “Open File” dialogs.

For most graphics tasks we use pure OpenGL. You will not need to write OpenGL calls yourself to fill requirements, but you might want to for extra credit. The assignment code should work on computers which support OpenGL 3.3 or later.

---

<sup>1</sup>Yes, we could use the very similar `std::format` if we were compiling on C++20. However, it is not supported on XCode except on the latest MacOS version, which we find a little limiting. We stick with C++17 for now.

Some parts of the base code use the [Google naming style](#). We do not judge your code based on style.

## 4 Detailed instructions

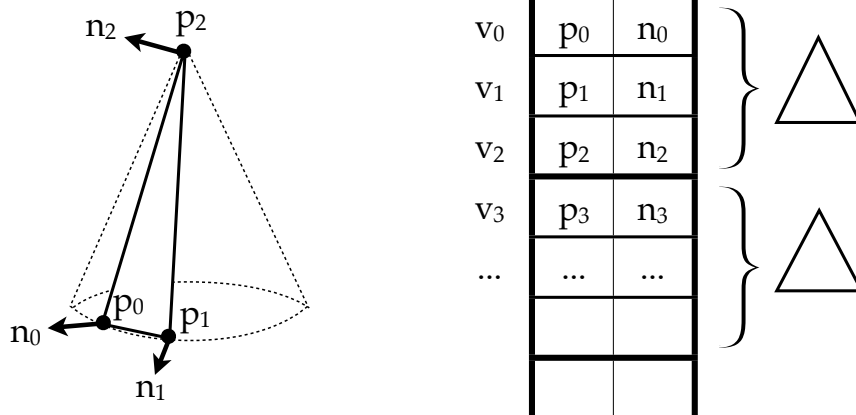
Requirements 1-3 do not depend on each other, so if you get stuck on one, you can start work on another. You can also work on R4, but to see its results you have to complete R3 first. There are more instructions and hints in the code comments, marked with `Rx` to make it easier to search for comments related to a particular requirement.

### R1 Moving an object (1 p)

Your job is to give the user a way to translate the object in space along any axis using the keyboard. (In `reference.exe`, arrow keys and `PgUp/PgDn` keys do this.) You'll need to

- add a new data member in the `App` class definition to store the current translation, and make sure to initialize it to the zero vector.
- change the model-to-world transformation (initially identity) within the `App::render` function to transform the model position correctly, and
- add code to the `App::handleKeypress` function to react to user keypresses.

### R2 Generating a simple cone and normals (3 p)



Your job is to complete the `loadGeneratedConeModel` function so it generates a model of an upright cone with its tip at the origin, a height of 1 and a radius of 0.25. You can leave the cone open at the bottom. Be sure to change to the correct scene by clicking on the GUI button to see your results.

The finished cone model will have 40 *faces* (triangles). The faces are represented as an array of 120 *vertices*; every set of three vertices in the array describe one face. Each vertex contains its position coordinates and its normal direction (see the definition of struct `Vertex` in `App.h`). For reference, you can see the vertex array data of a single triangle in `loadExampleModel` and the vertex array data of the reference plane (two triangles) in `reference_plane_data`.

For every face, you need to figure out the positions for its three vertices so that taken together the faces form a cone. When the vertex positions are correct, the cone will look right in the default shading mode.

Then, get the normal of the face from the vertex positions using cross product. You can use the same normal for all three vertices. This will result in a "flat shaded", not completely smooth look. When the normals are correct, the cone will look right in the directional light shading mode (press T or the GUI button to activate this) – compare with the reference, it's easy to get negative normals that are pointing inside the cone!

The function returns the data in a `std::vector` container. A vector in C++ is simply an array of objects of a specific type; `vector<int>` is an array of integers, `vector<Vertex>` is an array of vertices and so on. vectors are efficient and convenient; unless you have a reason to do otherwise, use a vector to store your data.

### R3 Converting mesh data for OpenGL viewing (3 p)

In R4 we are going to be loading 3D models in a compact, indexed representation. In this representation, every unique vertex position is stored only once in an array of positions, and likewise for vertex normals. Therefore a vertex can be represented as one index into the position array and one index into the normal array, and a face/triangle (which consists of three vertices) is represented as a set of six indices. You can see the indexed representation of a tetrahedron model in `loadIndexedDataModel`.

This kind of indexing scheme - where the data needed to draw a single vertex has to be gathered using more than one index - is not supported by graphics hardware. To efficiently render the mesh using the GPU we must convert to a simpler representation. What our OpenGL rendering code pushes to the GPU is a simple array of vertices where every vertex contains its position and normal directly, and every three subsequent vertices will be drawn as one triangle, as described in R2.

Your job is to fill in the `unpackIndexedData` function which converts indexed mesh data to vertex array form. This should take about 10 lines of code. When your code is correct, `loadIndexedDataModel` will start working and load the tetrahedron model as in `reference.exe`.

We need some datatype to hold the data of an individual face (six indices). We could use `vector<unsigned>`, but for clarity we use `array<unsigned, 6>` instead. The size of an array is spelled out in the code; it cannot grow or shrink like vector can. So the last function argument, which needs to represent all the faces, ends up as `vector<array<unsigned, 6>>` (an array of arrays of exactly 6 unsigned integers).

There is a bit more you should note about `unpackIndexedData`'s argument types, like `const vector<Vector3f>&`. In C++, function arguments are *passed by value* by default; if the argument type was defined as `vector<Vector3f>`, the entire vector would be copied every time the function is called. `&` denotes that the argument is instead *passed by reference*; the vector inside the function is the same object it was called with, not a copy. `const` indicates the function is not supposed to modify the argument, and gives you a compiler error if you try.

## R4 Loading a large mesh from file (3 p)

Before this we have used 3D models whose data is placed directly in the code or generated from scratch. This is impractical for all but the simplest models. To be able to use arbitrarily complex models, we will load them from separate files using the [Wavefront OBJ](#) format. OBJ is a widely used text-based file format that can describe a large variety of model data. In this assignment we will only read a part of that data and ignore the rest.

Your job is to fill in the missing parts of the `App::loadObjFileModel` function. It reads mesh data from a file into the same indexed representation as in R3, then uses the `unpackIndexedData` function you wrote in R3 to convert the data to viewable form.

In the `assets` folder there are three files: `sphere.obj`, `torus.obj` and `garg.obj`. These are the models we'll be testing your program with. Make sure you are able to load and view all three without crashing.

Let's take a closer look at `sphere.obj`. It's a big file, but it can be summarized as follows:

```
#This file uses ...
...
v 0.148778 -0.987688 -0.048341
v 0.126558 -0.987688 -0.091950
...
vn 0.252280 -0.951063 -0.178420
vn 0.295068 -0.951063 -0.091728
...
f 22/23/1 21/22/2 2/2/3
f 1/1/4 2/2/3 21/22/2
...
```

Each line of this file starts with a token followed by some arguments. The lines starting with `v` define vertex positions, the lines starting with `vn` define normals, and the lines starting with `f` define faces. There are other types of lines, and your code should ignore these.

Handling positions ("`v`") is straightforward; you just read their three numbers into a `Vector3f` and add it to positions. Then, do the same for the normals ("`vn`"), adding them into normals.

The faces ("`f`") are a little more complicated. Each face is defined using nine numbers in the following format: `a/b/c d/e/f g/h/i`. This defines a face with three vertices with position indices `a`, `d`, `g` and normal indices `c`, `f`, and `i` (you can ignore `b`, `e`, and `h` for this assignment). The general OBJ format allows faces with an arbitrary number of vertices; you only have to handle triangles. Read the indices into an `array<unsigned, 6>` and add it to faces.

One thing you have to watch out for is that the indices in OBJ format start from 1, but when we index into C++ containers we start counting from 0. Thus, you have to subtract one from every index you read.

C++ input and output is usually performed with *streams*. You can write some debug information in the console by using the standard *output stream*, `std::cout`, and the stream write operator `<<` like so:

```
cout << "The number is: " << 123 << endl;
```

In this requirement, we open an *input stream* that represents the contents of a disk file (ifstream). We use `getline()` to read one line at a time into the string `line`. We then represent that string as an `istringstream`, which is also an input stream. That allows us to use the stream read operator `>>` to read individual pieces of data from the string. (Note that `istringstream`'s operator `>>` uses whitespace in the string to tell where the next piece to be read begins and ends. The `/` characters between numbers in the file would confuse `>>`, so we must replace them with whitespace before creating the `istringstream`.)

```
ifstream input(filename, ios::in);
string line;
while(getline(input, line)) {
    istringstream iss(line);
    int i; float f; string s;
    iss >> i >> f >> s;
    ...
}
```

## 5 Extra credit

Here are some ideas that might spice up your project. The amount of extra credit given will depend on the difficulty of the task and the quality of your implementation. Feel free to suggest your own extra credit ideas!

We always recommend some particular extras that we think would best round out your knowledge. These will range from easy to medium difficulty. The base code will usually be written to accommodate the recommended extras, and may already contain some inert code and/or comments to help you out with them.

### Especially recommended

- **Version control (1 p)**

Use the git version control running on [Aalto GitLab](#) to store your assignment code. It protects your work from accidental destruction, and allows you to return to a working version of your code if you end up writing a bug and can't find it. Using a network repository will also help you work on multiple computers. *But make sure the repository you use to store assignment code is private!* We supply a .gitignore file with every assignment that will help you ignore useless files while using git.

While this has nothing to do with computer graphics as such, you'll get a bonus point as an incentive because we think using a version control system will improve the rest of your work. We strongly recommend you use Aalto GitLab, which is where you got the starter code by forking in the first place. To claim your bonus point, submit a log of your repository history or a screenshot of it. Adding `> logfile.txt` will redirect the log output straight to a file.

```
git log --graph --all --format=format:"(%h) %ci <%ce>%n%s"
```

- **Rotate and scale transforms (1 p)**

In addition to the object translation in requirement R1, allow the user to scale the current object (non-uniformly) along  $x$  axis, and to rotate the object around  $y$  axis. Pay attention to the order of the transforms so the result makes sense; translating or rotating the object shouldn't change its size.

- **Transforming normals in vertex shader (1 p)** (*requires rotate and scale transforms*)

As you rotate the object in light shaded mode, you'll see the shading does not change, as if the light was stuck to the object being rotated. You have uncovered a bug; obviously the assistant who wrote the vertex shader forgot to transform the vertex normal before it's used to do the shading calculation. Write the missing GLSL shader code in the vertex shader to fix the bug. For a full point, make sure shading also looks correct with uneven scaling (stretch the torus model to test this). **Harder alternative in Medium section.**

- **Better camera (up to 3 p)**

The base code has a very limited camera. Write your own camera matrix to replace the existing one in `App::render`, and give your camera different/better functionality. For example, you could allow freely moving and rotating the camera, or centering the camera around the object's position instead of the origin. See comments in `App::handleMouseMove` about adding mouse control. A working [virtual trackball](#) implementation is one way to get full 3 points.



## Easy

- **Animation (0.5 p)**

In `reference.exe`, you can press `r` to start and stop camera rotation. Implement this in your code. See the `Timer` class in the shared sources. You can use it for checking how much time has passed since the last frame and act accordingly. You can animate something else instead of the camera if you like, such as a transformation for the object.

- **Viewport and perspective (1p)**

Allow the user to choose the field of view in the `x` direction and adjust the other to match. Use an ImGui slider to allow the user to control it.

## Medium

- **Efficiently transforming normals in vertex shader (2 p)** (*requires rotate and scale transforms*)

This is an alternative for the Recommended extra. You might want to do it before attempting this.

Again, you must transform normals correctly also when the object is being scaled along one axis. Just modifying the GLSL code to do this results in an expensive calculation being carried out for every vertex. Since that calculation is the exact same at every vertex, we don't want to do it in the vertex shader. Your job is to calculate the transform matrix for normals in the C++ code, add a new *uniform variable* (a shader variable common to all vertices) for passing the matrix into the shader, and use it inside the shader code to transform the normal. Check out how the existing uniforms work.

- **Add support for loading another file format (up to 4 p)**

Allow the user to load models from another 3D model format such as [PLY](#). Include a model for testing your code.

## Hard

- **Mesh simplification (? p)**

Large meshes are quite difficult to draw and process. For interactive applications, such as video games, it's often desirable to simplify meshes as much as possible without sacrificing too much quality. Implement a mesh simplification method, such as the one described in [Surface Simplification Using Quadric Error Metrics](#) (Garland and Heckbert, SIGGRAPH 97).

- **Rendering Signed Distance Fields (? p)**

While meshes are the typical representation actually drawn using graphics hardware, nothing says that the vertex positions and connectivity have to be specified directly "by hand". *Signed distance fields* are a common volumetric geometry representation that can be converted to meshes using, for instance, the "Marching Cubes" algorithm. Implement signed distance fields, including conversion to a mesh for display. The representation allows you to do interesting things that are hard to do with meshes directly, such as enlarging or shrinking surfaces so that they do not intersect themselves. Demonstrate something like that. Be sure to include any assets needed by your code in your submission package.

## 6 Submission

- Make sure your code compiles and runs both in Release and Debug modes on Windows with Visual Studio 2022, preferably in the [VDI](#) VMs. You may develop on your own platform, but the TAs will grade the solutions on Windows.
- Comment out any functionality that is so buggy it would prevent us seeing the good parts. Use the `reference.exe` to your advantage: does your solution look the same? Remember that crashing code in Debug mode is a sign of problems.
- Check that your `README.txt` (which you hopefully have been updating throughout your work) accurately describes the final state of your code. Fill in whatever else is missing, including any feedback you want to share. We were not kidding when we said we prefer brutally honest feedback.
- Package all your code, project and CMake+vcpkg configuration files, required to build your submission, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive. *Do not* include your build subdirectory or the reference executable. Beware of large hidden `.git`-folders if you use version control - those should not be included.
- Sanity check: look inside your ZIP archive. Are the files there? Test and see that it actually works: unpack the archive into another folder, run the CMake configuration, and see that you can still compile and run the code.
- If you experience a system failure that leaves you unable to submit your work to MyCourses on time, prepare the submission package as usual, then compute its MD5 hash, and email the hash to us at `cs-c3100@aalto.fi` before the deadline. Then make the package that matches the MD5 hash available to us e.g. using `filesender.funet.fi`. *This is only to be done when truly necessary; it's extra work for the TAs.*

**Submit your archive in MyCourses folder "Assignment 1: Introduction".**