ME-C3100 Computer Graphics, Fall 2018

Lehtinen / Aarnio, Kemppinen, Ollikainen

Assignment 1: Introduction

Due September 26th at 23:59.

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)**

1 Getting Started

- First open the file `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 `example.exe` program. Your program will work like this when it fills the requirements. (The example also has minor features that are not required but will give you extra credit.)

- Compile and run the exercise code. Double click the file `assignment.sln`; this should open the code in Visual Studio 2017. Right click the 'assignment' project on the left, and choose 'Set as StartUp Project'. Navigate to '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 example.

- At least skim the rest of this PDF before you start coding!

2 Tour of the environment and the code

2.1 Environment and tools

The Aalto classrooms should have Visual Studio 2017 installed. For working from home, you can get the free VS2017 Community for Windows Desktop (link). This requires signing in on a Microsoft account but it's free.

The `assignment.sln` file you opened is a VS2017 *solution*, essentially a workspace. It contains two *projects*, `base` and `framework`. The `base` project contains the files specific to the current assignment; all files you need to modify are there. You should never need to modify the project settings or add new files to complete the requirements, but you may do so. `framework` contains supporting code that stays the same between assignments.

There are two *build configurations* in the Visual Studio solution you can choose from: Debug and Release. 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!

2.2 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 – 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`.

To handle some graphics and UI tasks, we use a utility framework from Nvidia Research which sits in the C++ namespace `FW`. We try to cover the crucial `FW` functions and classes either by example in the supplied assignment code or other course materials, so you don't need to dig deep into `FW` code yourself; it isn't documented and can be cryptic. The only part of `FW` you will need to use constantly is the math library found in `Math.hpp`, such as `FW::Vec3f` vectors and `FW::Mat4f` matrices.

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 better.

Some parts of the base code use the Google naming style. We do not judge your code based on style.
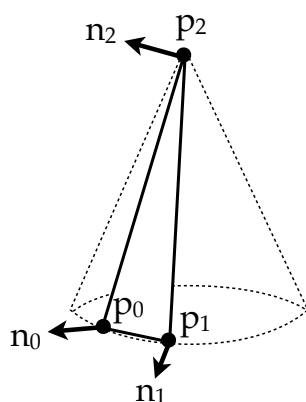
3 Detailed instructions

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 `example.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

- 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::handleEvent` function to react to user keypresses.

R2 Generating a simple cone and normals (3 p)

Your job is to complete the `loadUserGeneratedModel` 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.

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.hpp`). 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.

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. `vector`s 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 `example.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<Vec3f>&`. In C++, function arguments are *passed by value* by default; if the argument type was defined as `vector<Vec3f>`, 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 `Vec3f` 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;
  ...
}
```

4 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 a version control system such as `git`, `hg` or `svn` 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 any repository you use to store assignment code is private!* Bitbucket is one provider of free private `git` and `hg` repositories. 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 VCS will improve the rest of your work. To claim your bonus point, submit a log of your repository history or a screenshot of it. If you use one of the above VCSs, use the commands below to create the log. (Adding `> logfile.txt` will redirect the log output straight to a file.)

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

- **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 assistent 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::handleEvent` about adding mouse control. A working virtual trackball implementation is one way to get full 3 points.

Easy

- **Animation (0.5 p)**
  In `example.exe`, you can press `r` to start and stop camera rotation. Implement this in your code. See the comments in `App.hpp` concerning the `FW::Timer` class. Note that `App::handleEvent` is called periodically, so you can check there how much time has passed and act accordingly. You can animate something else instead of the camera if you like, such as a transformation for the object.

- **Viewport correction (0.5 p)**
  If you resize the application window to a non-square aspect ratio, you'll see a stretched image. Behind the scenes, `FW` is automatically adjusting the OpenGL viewport to match the size of the window. Our rendering code keeps rendering the same geometry, which then looks wrong. Fix this by manually setting the OpenGL viewport at the beginning of `App::render`. Make it a square in the middle of the window, and as large as you can. You can check the size of the window in pixels with `Window::getSize`. OpenGL viewport coordinates are set with `glViewport`. **Harder alternative in Medium section.**

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.

- **Viewport and perspective (1-2 p)**
  Instead of setting the viewport to a square in the middle of the window like in the easy extra, react to changes in window size by adjusting the perspective matrix so you render an appropriate part of the scene to fill the window without distortion. For full points, allow the user to choose the field of view in one direction and adjust the other to match.

- **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).

5   Submission

- Make sure your code compiles and runs both in Release and Debug modes on Visual Studio 2017. Comment out any functionality that is so buggy it would prevent us seeing the good parts. 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, `README.txt` and any screenshots, logs or other files you want to share into a ZIP archive.

- Sanity check: look inside your ZIP archive. Are the files there? (Better yet, unpack the archive into another folder, and see if you can still open the solution, compile the code and run.)

Submit your archive in MyCourses folder "Assignment 1".