Computer Graphics
Lab instructions

Mathias Broxvall, November 2013

General instructions

This text together with some files on the course web page describes the labs in the course Computer Graphics, given at Örebro University Autumn 2013.

Platform

The labs are primarily intended to be performed under Windows in the University computer halls. However, since OpenGL and related libraries are freely available also for most other platforms, you are encouraged to also use your own laptops or other computers to perform the labs under Windows, Linux or MacOS. If you want to use another platform than the University classrooms you will find some useful information on the course web page, however note that it is your own responsibility of making it work in case you choose alternative platforms.

Program code

In order to solve the exercises you will have to use or build upon some existing code. All the files needed to complete the labs can be found at the course web page and you can download it to your computer during the labs.

`http://www.aass.oru.se/˜mbl/datorgrafik`

Preparations

In order to benefit properly from the laborations you are expected to prepare well before each lab. Remember that this is for your own benefit, and unprepared students may be denied assistance. These preparations include:

1. To have read the exercises and any other material such as lab skeletons, or other source files and instructions. If any of the exercise assignments or questions are unclear you should be able to ask the instructor in the beginning of the laboration.

2. Go through the lecture notes covering the corresponding area.

3. You should have thought through how the exercises can be solved. This can be done, for instance, by sketching up a rough structure (the design) of your program and to lookup built-in functions in the OpenGL 3.3 and GLSL standards that may be used. Please see the official documentation that can be reached either one of:

   `http://www.opengl.org`

   `http://www.khronos.org`

4. Considered what can possibly go wrong in your program and how you can fix it. Also make plans for how you can test different parts of your program while writing it, and finally how you can test the whole program. *Note that it can be very hard to debug OpenGL and GLSL code unless you think it through carefully.*

Examination

For the labs you may work individually or in pairs. Although you are encouraged to discuss different solutions with your classmates you are not allowed to copy code.

All labs should be handed in with a written report latest at the date given in each lab that should be sent *electronically* as a PDF document by email.

Additionally, observe that the few assisted labs in the course means that you *will have to work outside the scheduled time* in order to finish in time, and that you are expected to be able to handle the programming environment and to debug your own code.

The examination of each lab consists of **two obligatory parts**:

- A lab report describing the program and your solution as well as the *important parts of the code for each exercise*. The report should *describe* and *motivate* the solutions you have used, the intentions behind your solution, a description and parameters for each function you have written.

  Additionally, you should append the complete code for the final parts of each lab with your report.

  Control that your report and code are correct, can be run without errors and is well formed and commented before you hand it in. Code that does not satisfy these requirements are being returned without corrections and count as one missed examination chance.

- An demonstration of the exercise where the instructor will give a number of questions related to the laboration. If you work in pairs, then both must be able to answer any questions about any part of the lab. This is to be done before you send the report to the teacher.

To pass the laboration part of the course you are required to hand in and pass each laboration in time. To pass the labs, the following is required:

1. Each lab must be demonstrated before they are handed in. They can be demonstrated either during a lab occassion or by booking an appointment by email. Note that if you choose the later your options for times to demonstrate may be limited.
2. The reports must be handed in *at latest* 13:30 the day printed on the exercise. The labs are corrected within one week and returned at the next scheduled lab occassion. The results of the correction can be either *pass*, *completion* or *fail*.
3. For a lab to be corrected it must be a *reasonable attempt* at solving the exercise, handed in in time, and have *well commented* code. Otherwise it will automatically count as a completion.
4. If you have received *completion* on a lab you must hand in the completed lab report *within two weeks*. The correction of this report will be done within a week and will give either *pass* or *fail*. **Observe that you cannot complete a lab more than once!**

The report must satisfy at least the following requirements:

- All questions in the exercise must be correctly answered and motivated. Only repeating the question or providing a factually correct yes/no answer does not count as answering it.
- The solutions to exercises must be briefly described. The code for the *actual* (but not all past exercises) problems must be included, correctly indented and well commented.

If you have used any code from other sources than the lab skeletons (eg. Internet, books etc.) you have to clearly mark this in the source code. Not doing so counts as attempted cheating and you will be reported to the disciplinary board. Depending on the amount of borrowed code, the instructor may fail the lab – therefore always discuss with the instructor before using any code that you have not written yourself.

Examples of code that is ok to use is the glm library (from exercise 2.3 and forward), external libraries for loading images and/or model descriptions. Example of code that is NOT allowed is external libraries for rendering the models, shaders copied from other sources etc.

Due to the limited instructor time, all labs must be completed and passed within the allotted time of the course, otherwise you will have to resubmitt the labs the following year.

Finally, a short clarification of the expected work. Since this is a half-time course you are expected to spend 20 hours/week on this course. This entails roughly 4 hours of lectures and 4 hours of preparations before the lectures, 3 hours of teacher assisted laborations and 9 hours of preparations for the labs and labs done on your own time.

Examiner: Dr. Mathias Broxvall, mathias.broxvall@oru.se

Compiling the labs under Windows

In order to compile and run OpenGL programs under Windows you will need to link the program executables against the OpenGL libraries. In addition, we will use the SDL library to provide a light weight environment giving an OpenGL context and other primitives for handling keyboard input, generation of sound etc. This will require that also the SDL library is downloaded and linked against the application.

To create a lab project, start by launching Microsoft Visual Studio and creating a new `Win32 > Win32 Console Application` project. Give it a suitable name, press next and click on the checkbox that it should be an empty project. Continue by downloading the corresponding lab skeleton file from the course webpage (eg. lab-1.1.zip).

Unpack the file and move all source code (`.c`,`.cc`) and header (`.h`) files to the folder where your projects sources will be. If your project is named `foo` and is saved under `M:/labs/foo` then this is usually under `M:/labs/foo/foo/`.

Add all these source code and header files to the project by going to `project > source files > add > existing items` and selecting them.

If you are developing on a computer that does not have libSDL and SDL_image installed you need to download these. In the classroom open the file `windows.zip` and copy the content of the `windows/dev` folder to where your source code files are. When you have a ready application to be run from another folder or another computer you need to place the content of the `windows/bin` folder in the same place as the executable.

Before you can compile the program you need to do one more thing, to instruct the linker to use the libraries you just copied. Go to `project > properties` and then to `Configuration Properties > Linker > Input`. Add the following line to the *Additional dependencies*:

```
opengl32.lib glu32.lib SDL.lib SDL_image.lib SDLmain.lib
glew32.lib
```

Now you can compile the program by pressing F7 and to test run the program through the debugger by pressing F5.

Compiling the labs under Linux

To compile the labs under linux you need to first install the SDL library and header files. Under Ubuntu you can do this by installing the packages `libsdl-dev` and `libsdl-image1.2-dev`. After this you should be able to just run `make && ./lab-1_1`.

Lab 1 - An introduction to OpenGL and GLSL

In this first lab you will first shortly get familiar with the operating system and your development environment and then solve a number of shorter OpenGL exercises of increasing difficulty. For a full specification of the OpenGL functions that are available to be used, please see `www.opengl.org`. Note that there is a list of useful OpenGL and GLU functions towards the end of this lab. Please look up these functions online and read what they does and how they can be used in your code.

Exercise 1.1 Your first OpenGL/GLSL program(s).
Download, unpack and compile lab 1.1 from the course webpage. Launch the application. Congratulations, you now have your first OpenGL application running – showing a very interesting dark yellow screen that does nothing. Quit the application by pressing the escape key (or closing the window).

Your job now is to make the lab a bit more interesting.

If you take a look in the code you will note that there are a function `World::doRedraw` that is called periodically. This function already contains some OpenGL commands that launches two pieces of a *shader program* drawing a rectangle (consisting of two triangles) covering the full screen. The source code for the shader programs can be found in `vertexShader.vs` and `fragmentShader.fs` and corresponds to the *vertex shader* and *fragment shader*.

Start by opening `fragmentShader.fs` and try to change the colour that is drawn to the screen. This is done by changing the colour that is assigned to `fragmentColour`. You can for instance add the `position` variable to the `vertexColour` to get a gradient over the full screen. Note that you can make changes to the sourcecode of the shader files while the program is running, and recompile it on the fly by pressing the 'l' key.

Next, you should continue by extending the *doRedraw* function of your main application so that it will draw some more interesting objects. Start by adding two more triangles looking like a rectangle that covers a part of the center of the screen. You can do this by extending the list of vertices that are given to OpenGL and by extending the list of graphic primitives and corresponding vertex indices that are given for the drawing command.

The rectangle appearing from these two triangles should have the pure colours *red, green, blue, yellow* in the four different corners and *there should be no visible edge* between the triangles that make up the rectangle. You should use `glDrawElements` for this and extend the list of vertices (and colours) by exactly four and the list of triangles (indices) by two..

Take a screenshot of your application (this should be included in your lab report).

Continue by changing the direction in which your rectangle is subdivided (eg. from top-right to bottom-left), but keep the colours in the same positions on the screen (thus you will need to change the order in which you give the colours to OpenGL). Ideally this should look identical to your previous screenshot, *does it look the same?*.

Take yet another screenshot of your application and compare it with the first.

Finally, change your code to use `glDrawArrays` to draw both of the rectangles now on the screen. Take a screenshot and compare it to the two previous ones. What can you learn from this?

Question 1.2.
Explain *why* there is a difference between the interpolated colours in the different screenshots above.

Exercise 1.3 More GLSL programming.
Start by changing your *vertex* shader so that the colours that are passed to the fragement shader corresponds to a position on a colour wheel. The position on the colour wheel is given by the following formula:

$$\text{rgb} = (0.5 + 0.5\sin(p), 0.5 + 0.5\cos(p), 1.0)$$

where p is a `phase`. Extend your shader to accept one input `float` variable `inPhase` and extend the `main.cc` code to pass new attriute values to this variable. This is done by adding one more array and passing it to OpenGL using `glVertexAttribPointer`.

Think carefully about what "index" to use for this new variable, also think about how many elements and dimensions that your array needs to contain.

Pass the value 0.0 as the "phase" to vertices on one edge of the screen and 6.0 to the vertices on the other side. Do you get the result that you expected? Take a screenshot to be included in the report.

Continue by changing your code so that instead of drawing just one rectangle covering the whole screen you draw N rectangles each covering a slice of the screen. Eg.

```
for n=0 to N
  x_start = n/N*2.0 - 1.0
  x_stop = (n+1)/N*2.0 - 1.0
  phase_start = 6.0 * n/N
  phase_stop = 6.0 * (n+1)/N
```

Compare what happens if you use different values for N. (Try both $N = 2$, $N = 10$ or $N = 100$).

Next, continue by adding an `out` variable from your vertex shader called `phase` and pass the `inPhase` argument to this variable directly. *This means that we will perform linear interpolation of the used phase instead of the used colour.*

Change your fragment shader code to perform the colour wheel computation given above using this value for phase. Take a screenshot to be included in the report and compare this to the appearance you got when the colours where computed and interpolated in the vertex shader.

Finally, add a *uniform* float variable called `time` to both of your shaders and augment the colour computations to instead be:

$$\text{rgb} = (0.5 + 0.5\sin(p + t), 0.5 + 0.5\cos(p + t), 1.0)$$

where t is your time variable. Add a corresponding member variable *worldTime* to your world class, let it be updated by the *tick* function and augment the *doRedraw* function so that this `uniform` variable is passed to your shaders. You can do this by adding the following just *before* the drawing commands.

```
GLuint loc = glGetUniformLocation(r->shaderProgram,"time");
glUniform1f(loc,myTime);
```

Note that we have here not used the layout qualifier. The same technique of querying for the location of a variable can be used both for uniform variables and for attribute variables.

If you have done this correctly you should now have an animation where the colours are moving over the screen. Compare the animation quality between the fragment shader and the vertex shader based code (for different values of N).

Question 1.4.
Explain *why* there is a difference between the interpolated colours in the different screenshots above.

Explain also the trade-off in computational burden on the GPU versus the CPU as you choose which one of the two shaders to use.

Exercise 1.5 Additional drawing primitives.
Read online to see how the drawing primitives `POINTS, LINES, LINE_LOOP` and `TRIANGLE_STRIP` is working.

Augment your code from the previous exercises to draw atleast one of each of the primitives above. Include a screenshot that shows all these priitives in the lab report.

Note that you are still NOT allowed to use depracated functions such as glBegin or glVertex.

Question 1.6.
Download the *OpenGL Shading Language 4.40 Specification* from khronos.org and lookup the build-in GLSL function “mix”. Explain what this function does and give an example of when you may want to use it. Also explain why it is better to use this function rather than writing it yourself.

Examination

Recommended time: One week
Latest examination date: 2013-01-16

Interesting Functions

Use the world wide web to look up what these functions do, you *will* find them usefull in your exercises. If you are brave you may also read the OpenGL 3.3 (or 4.2) and the GLSL specification as downloadable from `opengl.org`.

- The qualifiers `in, out, layout` as used in GLSL shaders.
- What so called `uniform` variables are in GLSL.
- The built-in datatypes `vec3, vec4` used in GLSL shaders.
- `glVertexAttribPointer` and the `layout` qualifier in (vertex) shader programs. Note especially the *index* passed to the vertex attrib functions.
- `glDrawArrays, glDrawElements`

Deprecated functions

The following functions are marked as *depracted* in OpenGL 3.x and have been removed from OpenGL 4.0 and onwards. You should *not* use any of these functions in your OpenGL programs and you will not pass the lab if you do.

- Immediate mode drawing functions, eg: `glBegin, glEnd, glVertex`
- Fixed functionality matrix operations, eg: `glMatrixMode, glLoadIdentify, glTransform, glRotate, ...`
- Fixed functionality lighting, eg: `glLight*`
- Fixed functionality per vertex attributes, eg: `glColor, glNormal`

Lab 2

3D graphics, lighting and textures

In this first part of lab 2 you will experiment with how to draw 3D graphics with perspective and modelling transformations. You will learn how to perform lighting computations, give normals for the objects to be drawn and separate the *modelview* matrix from the *projection* matrix. Finally you will load some image files to be used as a 2D texture that is draped over the object you draw and experiment with letting this texture modify the different properties used in the lighting computations.

As you perform more and more advanced math operations you will need to perform operations such as matrix multiplications, matrix inversions and some other operations that might be hard for you to implement by hand.

To make the code more readable, and easier to use it is therefore recommended to use GLM (OpenGL Mathematics). This is an OpenSource headers-only implementation of the GLSL primitives (`vec2, vec3, mat4, ...`) in C++ that is released under an MIT license. You will find it at `http://glm.g-truc.net/`.

Go to this website and read the GLM manual before the lab

You will need to download these headers and include them in your project files (starting with exercise 2.3).

Exercise 2.1 - Performing perspective transformations.
Download, unpack and compile lab 2.1 from the course webpage. This will draw one quad over the full screen, but with different z-values on the left and the right edge – but this is not yet visible since we do not have any perspective transformations.

Start by adding a `uniform mat4 projectionMatrix` to your vertex shader code. This will become the matrix to be used for your perspective transformation.

Create a corresponding array consisting of 4x4 `GLfloat`'s on the CPU side and initialize this matrix to:

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -a & -b \\ 0.0 & 0.0 & -1 & 0.0 \end{pmatrix}$$

where a, b are computed from the *near* and *far* clipping planes $n = 1, f = 10$ by:

$$a = (f + n)/(f - n)$$

$$b = 2fn/(f - n)$$

Pass this matrix into the corresponding projection matrix above using the `glUniformMatrix4fv` function. Take care to note if you have specified your matrix in *row major* order or *column major* order.

Finally, perform the projection by multiplying this matrix in front of the position variable in your vertex shader. Note that GLSL has built in matrix-vector and matrix-matrix multiplications.

However, since the quad above ranging from $z = -1$ to $z = 1$ lies ahead of your near clipping plane (at $z = -1$) you need to first move the quad into the right position. In the next exercise you will do this with a matrix – but for now you can simply subtract 4 from `inPosition.z` before you multiply it by the projection matrix.

If you have done everything correctly you should now see the quad above with a perspective projection.

Note. This exercise does not need to be included in the report or demonstrated.

Exercise 2.2 - Performing 3D world transformations.
In this exercise you will continue from the previous exercise and use a *modelview* matrix to perform 3D transformations on your scene.

Continue from the code from exercise 2.1 and start by introducting a new `modelviewMatrix` in your vertex shader.

Create again a corresponding array on the CPU side and initialize it to the following:

$$\begin{pmatrix} \cos\alpha & 0.0 & \sin\alpha & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ -\sin\alpha & 0.0 & \cos\alpha & \Delta z \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix}$$

and use $\alpha = \text{time}$ and $\Delta z = -4.0$. This transformation matrix corresponds to a rotation around the y-axis by α radians followed by a translation of Δz units along the z-axis.

Pass this array to your vertex shader and multiply it *in the proper order* with your projection matrix and with passed input vertices. *Note: you should no longer manually change the z-value of the input vertices to offset it along the z-axis since this is done by the matrix above.*

Continue by extending your CPU application to pass in 6 quads corresponding to the 6 different sides of a cube that is ± 1 units on each side. Let each side have a different colour. *Note: you should not try to re-use the vertices for the different sides on this exercise. You will need to give it 6x4 vertices.*

Your result should now be a rotating cube that has a flat uniform colour on each side (and the top/bottom sides are not visible). To make sure that you have added the top/bottom sides correctly try to introduce a corresponding offset to the y variable in your modelling matrix above.

Note. This exercise does not need to be included in the report or demonstrated.

Exercise 2.3 - Using GLM for your math operations.
Continue on your exercise above and change your modelview matrix on the CPU side to be a `glm::mat4` value (if you have not already done so). You will need to include `glm/glm.hpp` to do this.

Remove your hand-coded transformation and translation matrix and compute instead the modelview matrix as the product of one `glm::translate` that moves the object along the z-axis and one `glm::rotate` that rotates it around the y-axis as in the previous exercise.

Next, add one more rotation – this time by time $* 0.4$ along the x-axis. Merge this with your modelview matrix above and pass only the composite matrix along to your vertex shader.

Your result should now be a rotating cube that has a flat uniform colour on each side and that rotates slowly along two axises at different speeds. This will let you see all of the cube given enough time.

This exercise *DOES* need to be reported and demonstrated.

Exercise 2.4 - Adding some lights to your scene.
In this exercise you will continue on your code above and add lighting computations to get a *"correct"* colour on each pixel of the cube.

Begin by adding a new vertex attribute `in vec4 inNormal` to your vertex shader. Add a new corresponding array on your CPU side and *give normals for each vertex in each quad* that is pointing out from the cube. Ie. for the quad that is facing the camera (at $z = -1$) you should have the normal $(0, 0, -1, 0)^t$. Note especially that you should have $w = 0$ in these normals – this is since the normals are not a *point* but rather a *direction*.

Next, continue by adding two output variables `position` and `normal` to your vertex shader. The `position` variable should be used to pass along the transformed 3D coordinate to the fragment shader, ie. without a perspective projection. The `normal` variable should be used to pass along the transformed 3D normal to the fragment shader.

Finally, create a new uniform variable `light0pos` in the fragment shader, initialize a corresponding variable to be $(2, 1, 0, 1)^t$ on the CPU side and pass it to the shaders. You are now ready to start computing colours.

Implement the phong lighting model on the GPU by the following formula:

$$C = C_a L_a + \sum_L \left(C_d L_d (\vec{N} \cdot \vec{V_L}) + C_s L_s (\vec{r_L} \cdot \vec{V_e})^f \right) \tag{1}$$

You can at this stage use the hardcoded constants:

$$C_a, C_d = (0.8, 0.8, 0.5)^t$$

$$C_s = (1.0, 1.0, 1.0)^t$$

$$L_a = (0.2, 0.2, 0.2)^t$$

$$L_d = (0.8, 0.8, 0.8)^t$$

$$L_s = (3.0, 3.0, 3.0)^t$$

$$f = 20.0$$

For N you should use your `normal` variable that is interpolated from the vertex shader – but you need to *normalize* it. This can be done by the built-in function `normalize(...)`.

For V_L (the vector to the light source) you should use the expression `light0pos - position` and *normalize* the result.

For V_e (the vector the eye) you should use just `position` since you know that the eye is in origo. Again, *normalize* the result.

Finally, for the reflection vector r_L you can use the built-in function `reflect`, or the expression below.

$$\vec{r_L} = 2(\vec{L} \cdot \vec{N})\vec{N} - \vec{L}$$

If you do these steps correctly you should now have a correctly lit object. **This exercise does not need to be reported and demonstrated.**

Exercise 2.5 - Texturing your objects.

In this exercise you should build on the code in your previous exercise 2.4 and add two textures that give the *ambient and diffuse* respectively the *specular* colour and intensity of the object.

Start by adding a `GLuint diffuseTexture` and `GLuint specularTexture` to your class `Resources`. Add the following two lines to the constructor of this class.

```
GLuint textures[2];
glGenTextures(2,&textures);
diffuseTexture=textures[0];
specularTexture=textures[1];
```

What this does is to create two new texture objects (with no actual content) that are references by the above two integers. Note that you could also have used to calls to `glGenTextures(1, ...)` instead of the above code if you wanted to.

Add the code to load some images to these textures. For this purpose you can use the function `loadTexture` that I have provided for you in `glUtils.cc`. Please take a look at this code and read it. The following code snipped should be added to your `Resources::reload` function.

```
loadTexture(diffuseTexture,"texture1.png", 1);
loadTexture(specularTexture,"texture2.png", 1);
```

Continue by adding the two new uniform variables `uniform sampler2D diffuseTexture, specularTexture` to your fragment shader. These variables are used as references to two different *texture units* on the graphics card. A texture unit is a dedicated hardware unit that can quickly read from and interpolate values from the texture memory in an efficient manner (and cache results between different invokations).

To let the fragment shader know which hardware unit to use you need to set it from the CPU. Use the CPU side of the code to set these two uniform variables to the values 0 respectively 1. Note that you use `glUniform1i` for this (you give sampler objects an integer number from 0...MAX_TEXTURE_UNITS).

Next, add the code to *bind* the two textures above to the first two available hardware texture units, numbered 0 and 1. You can do this by selecting the hardware texture unit to modify using `glActiveTexture(GL_TEXTURE0+i)` where i is the hardware unit to modify. (Nb. `GL_TEXTUREi` for any i is also defined as the above expression). After selecting the hardware unit to modify, bind the selected texture to it by using:

```
glBindTexture(GL_TEXTURE_2D,r->diffuseTexture};
glEnable(GL_TEXTURE_2D);
```

You are now almost ready to start performing texturing inside your fragmentshader. But first you need to decide which part of the texture images should correspond to each fragment that you are drawing. To do this, add one more `vec2` vertex attribute `inTextureCoordinate` to your vertex shader. Let the vertex shader pass along this coordinate unchanged to the fragment shader in a new `out` variable. Add the corresponding array on your CPU side to initialize these attributes and give *reasonable* values in the range $0 – 1$ for these texture coordinates so that the full textured image will appear on each side of your cube (after the next step).

Finally, you can add the fragment shader code that will actually fetch the data from the texture units and do something interesting with it. You can do this by adding:

```
vec4 Cd = texture(diffuseTexture, textureCoordinate);
vec4 Cs = texture(specularTexture, textureCoordinate);
```

Now, use the two values C_d and C_s above for you lighting computation.

You should now have textured cube where one texture is deciding the general colour of the cube and the second texture decides which parts of the cube have a specular reflection or not.

Try experimenting with the texture coordinates. What happens if you multiply the texture coordinates by a scalar constant (eg. 2) or by a vector constant (eg. `vec2(2.0, 1.0)`) before the texture fetch.

This exercise *DOES* need to be reported and demonstrated.

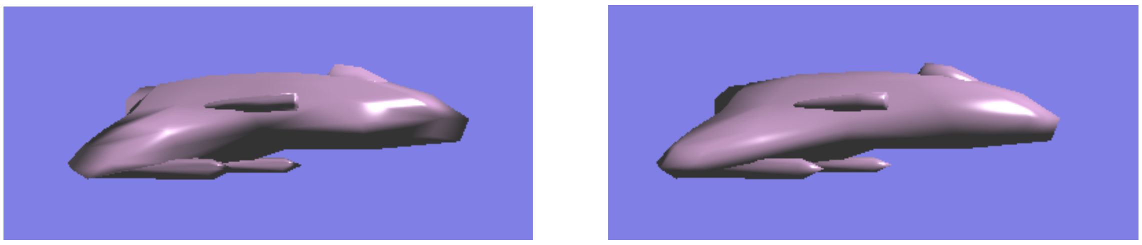


Figure 1: Fighter model file with flat normals (left) and with smooth normals (right).

Putting it all together

You will now continue by using what you have learned in this lab to make some more realistic scenes than drawing a mere cube. For this purpose you will use a simple 3D fileformat (AC3D) and a custom reading function that should be enough for your purposes.

Exercise 2.5 - Drawing an object.
Download the new `glUtil.cc` file from the course webpage and use it to load the example model file `fighter.ac` in the initialization of your resources. You can do this by first instancing the `AC3DObjectFactory` class and using the `loadAC3D` function. The reason why we are using an instanced factory class for this will become apparent later on.

The factory will return to you a hierarchy of AC3D object corresponding to the hierachy that appears in the file. Note that in the given example file, only the leaves of this hierachy of objects are interesting.

In your `doRedraw` code, loop over each such leaf and extract the vertices and surface indices from the object and render your object using `drawElements`. Note that you need here to allocate a new array of suitable dimensions and put the indices of the primitive surfaces there. Since the object contains different types of primities (triangles and quads), you need to extract separate sets of indices for both of them and to make two calls to the `drawElements` functions.

Change your fragment shader code from above so that it can draw an object *something* when not given a normal.

verify that you can see something on the screen with some form of silhouette.

Next, you need to create the normals for the object since these are not given as part of the AC3D file format.

Start by allocating space for *one normal per vertex* and initiallize each normal to be zero. Next, loop over all surfaces and compute the normal over each surface.

$$\vec{N}_s = \text{normalize}(\vec{P}_{s,1} - \vec{P}_{s,0}) \times \text{normalize}(\vec{P}_{s,2} - \vec{P}_{s,1})$$

where $\vec{P}_{s,i}$ are the vertex references by corner i of the corresponding surface s. Note that the fileformat standard of AC3D guarantees that the corners are given in a consistent order – so this normal computation will correspond to the geometrically correct value.

Next, set the normal of all of the vertices used each surface to this value. (Overwriting any old value, thus it is random which of the 3/4 possible normal is actually associated with each vertice that have 3/4 surfaces connected to it)[1]. If you have done this correctly it should now look similar to the left image in Figure 1.

Next, try to set the value of the normal for any vertice instead to be the *average* of the value of all the surfaces connected to it. *Hint:* this can be done easily by adding the XYZ parts to the normals in the loop over all surfaces, and in a second phase just normalizing the normals. Alternatively you can use the W part of the normals to store how many values have been added into them and make a division by W in the second phase (and resetting W to be zero at the time). Other possibilities of the second alternative is to weight the average differently for the different surface affecting a vertex, adding more (or less) contribution depending on the size of the surface.

If you have done this correctly it should now look similar to the right image in Figure 1. **This exercise does not need to be reported and demonstrated.**

Exercise 2.6 - Cleaning up the code.

So far you have mostly been doing the work of the lab in a single function, `doRedraw` that now (probably) start looking quite cluttered, inflexible and is inefficient since you do not reuse the computations on the objects. It is time to fix this and make a proper object-oriented approach.

Start by creating a new class `ModelObject` [2] that inherits from the `AC3DObject`. Add variables for storing the computed normals, indices and any other data that you need on a per *model* basis.

Create a new *factory* class `ModelObjectFactory` that inherits from the `AC3DObjectFactory` and implements a single method `makeObject` that returns a new `ModelObject` casted to look like a `AC3DObject`.

In the instantiation of your `Resources` class, change the used factory to be a ModelObject-Factory instead. Now when the AC3D file is loaded it will create a hierarchy of ModelObjects.

After the creation of ModelObjects, the `finalize` function is called on each object. Implement a new such function in your ModelObjects and use this function to allocate buffers for indices, normals and any other properties you'd like and to compute the "correct" values for them.

Change your `doRedraw` code to just use the right buffers and values from your ModelObjects.

Finally, make a new class `Fighter` that contains a *vec3 position* instance variable and a function `doRedraw`. Add two instances of this class to your *World* class (with slightly different values for the position variables).

Move the code for drawing the models of the fighter into the fighter class (which will get the actual data to be used from the ModelObject) and keep only the part of the code responsible for setting up camera perspective projections and camera position in your world class. For

[1]This is a hack, don't do this at home kids!

[2]you may always use whatever name you want for your functions and classes, but I'm giving suggestions to make the writing clearer

this purpose you need to pass the “current” camera modelview matrix in as a parameter to the `Fighter::doRedraw` function and to call it once for each fighter.

You may use the `glm::perspectiveFov` and `glm::lookAt` functions in

`glm/gtc/matrix_transform.hpp`

In the code for drawing the fighters, add a *translation* that moves it into the right position. Also draw only the “weapon*” object parts if the corresponding flags are set in the fighter. (Add new such flags to the class and give them different values for the two fighters).

You should now have two fighters that are both visible on the screen.

This exercise *DOES* need to be reported and demonstrated.

To pass this lab you should *not* have modified (in any significant way) **the core AC3D* classes**. You should not load the same AC3D file twice or perform the normals and indices computations more than one time.

Exercise 2.7 - Using buffer objects.

If you where to add lots of instances of your fighter class implemented in the previous exercise your code would start running slow due to a bottleneck between the CPU and GPU memory. This is since you are sending all the 3D data multiple times to the GPU on every frame – even though you only ever create/modify the model once.

In this exercise you will rectify this by using `buffer objects` that can be used to store vertex attributes as well as indices on the GPU.[3].

Start by allocating a *single* buffer *on the CPU* that can be used for storing your vertices and normals. You may either choose to store all the normals after all the vertices in this buffer, or to interleave them by storing a normal after every vertex position.

Draw your objects with the old fashined *CPU based* drawElements but using this buffer. If you choose to interleave the data you need to think about both the right starting address for passing in vertices respectively normals and to think about the *stride* argument given to `glVertexAttribPointer`.

All offsets and stride arguments are always given in bytes to OpenGL.

Next, continue by allocating a new buffer object using `glGenBuffers`, bind it as a `GL_ARRAY_BUFFER` object and copy the data from your buffer above into it. Change your calls to `glVertexAttribPointer` to be done *relative to the buffer*. Note that you need to compute the offsets of the corresponding vertex attributes and cast them into `(GLvoid *)` type.

Verify that your code works and give the same images as before. You have now done half the work in reducing the amount of data that is transfered.

Continue by allocating one or two buffer object(s) to store the index data for quads and triangles. Bind this object(s) as a `GL_ELEMENTS_ARRAY_BUFFER` and upload the corresponding indices to it using the `glBufferData` function.

In your draw routine, bind the correct elements array buffer and use the *relative address* in this buffer as the pointer to the data.

[3]note that there exists many different forms of buffer objects and you can store also other data such as uniform variables in them

Finally, you may (if you want) allocate a *vertex array* object and use this object to store the (a) bound vertex buffer, (b) vertexAttribPointer arguments and (c) enabled attributes. This reduces the number of function calls that are needed in the draw routine for an individual object to four: (a) binding the vertex array, (b) binding the index buffer(s) and (c) calling draw elements twice.

Question 2.1.
Try to think about and explain the difference in performance by storing the normals after the vertices or interleaved with them in your vertex buffer object. What do you expect to be better?

Examination

To pass the lab you must hand in a demonstrate and hand in a report covering exercise 2.3, 2.5, 2.6 and 2.7. Both the report and the demonstration must pass in order for you to pass the lab.
Recommended time: One week
Latest examination date: 2013-01-16

Interesting Functions

Use the world wide web to look up what these functions do:

- `glGenVertexArrays, glBindVertexArray`
- `glGenBuffers, glBindBuffer, glBufferData` and `GL_ARRAY_BUFFER` respectively `GL_ELEMENT_ARRAY_BUFFER`.
- `GL_STATIC_DRAW` in the context of `glBufferData`.

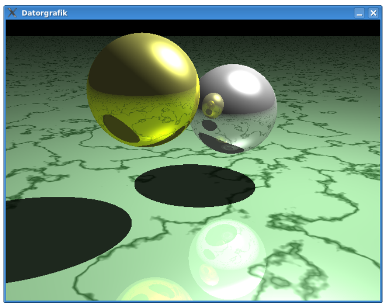


Figure 2: A simple realtime raytracer with procedural textures

Lab 3 - Raytracing

With the ever growing speed, and most importantly number of, processors in modern computers it is becoming more and more common to use raytracing both for offline and for online generation of images.

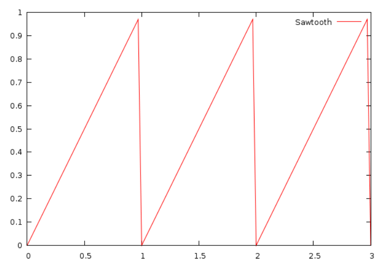
Some of the advantages of using raytracers is the simplicity with which advanced features such as true reflections, transparent materials with refractions, procedural textures, shadows and other features can be implemented. The major disadvantages, so far, is the large computational cost which still limits the use of raytracing significantly.

In this exercise you will test an alternative methods of generating graphics by using and extending a small realtime raytracer.

Start by downloading and compiling `lab-3.zip` from the course web page. To properly compile this program under Microsoft Visual Studio you still need to add the content of `windows.zip` since we use SDL for the window managment, but note that OpenGL is not used within this lab. Order to use all available processor cores on the machines, you need to also enable *OpenMP* which is a simple mechanism built-in to many compilers in order to utilize multiple threads for computations. To do this you need the following settings under *project properties*:

- Configuration: `Active(release)`. This is important for optimizing the speed, since otherwise the program will run at only a tenth of the proper speed.
- Linker → Input → Additional Dependencies: `SDL.lib SDLmain.lib`
- C/C++ → Optimization: `Maximize speed (/O2)`
- C/C++ → Language → OpenMP Support: `Yes (/openmp)`

When you now compile and run the program you will see a simple scene containing two reflective spheres bouncing on a glossy marbled floor, see Figure 2. This scene is raytraced in realtime and will use as many processors as are available in the computer and can achive approximately 20-30fps on the Intel Core2 CPU's in the classrooms. By holding down the left mouse button and dragging the mouse over the window you can rotate the camera.



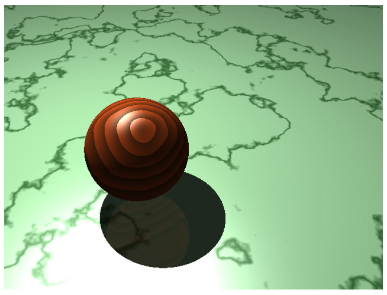


Figure 3: A sawtooth function (left) used to implement wood (right) by varying the blending degree between two materials depending on the radial distance.

Clicking the mouse enables a debugging output that is printed to the console - this will come in handy for you since you've just disabled all other debugging options. See inside eg, plane.cc for examples of how to do these printouts.

Exercise 3.1 - create a new material

To determine the arguments used during the lighting computations for each pixel the raytracer requests the lighting properties of the corresponding object at the 3D point corresponding to this pixel. This is done by a call to the `getLightingProperty` function which is then passed to the *material* which the object have been given or inherited from.

Your task in this exercise is to create a new type of material. There already exists examples for uniformly coloured phong shaded materials (see class SimpleMaterial) and for materials derived from a perlin noise function mapped by linear interpolation to corresponding phong shading properties (see class MaterialMap).

The material that you should create in this exercise is a wood material. This can be accomplished by using two sets of lighting parameters, light wood and dark wood, and by blending between the two of them so that the material will form concentric circles of light and dark wood when viewed down the Z-axis.

The wood should start out dark and grow smoothly lighter, drop sharply down to dark and begin going to light again (see Figure 3).

Ie. first compute the distance to the Z-axis:

$$r = \sqrt{p_x^2 + p_y^2}$$

Next, compute the blending factor to use by using a *sawtooth* function on $r * c$ for some suitable constant c. The easiest way to implement a sawtooth function is to use the `fmod` operator from `math.h`, ie:

$$b = \text{fmod}(r * c, 1.0)$$

Finally, to compute the final colour perform blending using b, ie. use:

$$(1 - b) * C_{\text{dark}} + b * C_{\text{light}}$$

Apply this new material to one of the spheres and view it from different angles. Does the result look good? What happens when you change the constant c above? What are good lighting properties to use for the light (eg. no specular reflection, light brown diffuse/ambient colour) and dark parts of the wood?

As you may notice, the result is unrealistic since real wood does not have the same thickness for each year ring. Try to modify your expression so that the material has smaller rings further in, and so that there is a degree of random fluctuations from year to year. For the later, you may want to use the one dimensional noise functions given in `noise.h`.

Exercise 3.2 - create a new primitive object

The next exercise requires a bit more mathematics to solve. In this exercise you will add another type of primitive object, an infinitly large cone, to the raytracer. The basic definition of this cone is the following parametric function:

$$f(\vec{p}) = \vec{p}_x^2 + \vec{p}_y^2 - \vec{p}_z^2$$

And the surface S of the cone is given by

$$S = \{\vec{p} : f(\vec{p}) = 0\}$$

This parametric definition will yield an inifitly large cone with it's apex at Origo and growing in radius (x/y) as we step up the positive z axis. Due to the symmetry of the expression, it will also yield another cone meeting the first cone in origo and growing in radius as we step down the negative z axis.

In order to implement this object in the raytracer you need to create a new object class and provide the three functions `lineTest`, `getNormal`, and `isInside` for it.

Exercise 3.2.1

The first of these functions, lineTest, requires a bit of math. To implement it you must analytically derive an expression for when a line starting at *origin* $\vec{o}$ with the given *direction* $\vec{d}$ intersects the object. The result should be the smallest α such that:

$$\vec{o} + \alpha\vec{d} \in S \quad \wedge 0 < \alpha \leq \max$$

where max is the *maximum distance* at which we are interested in intersections. If no such intersection exists then return max instead.

This becomes the same as solving the quadric equation:

$$f(\vec{o} + \alpha\vec{d}) = 0$$

Which can be simplified to:

$$(\vec{o}_x + \alpha\vec{d}_x)^2 + (\vec{o}_y + \alpha\vec{d}_y)^2 - (\vec{o}_z + \alpha\vec{d}_z)^2 = 0$$

Use this expression to derive an analytical expression for the *up to two* solutions that exists for α and return the first positive such solution, if any.

In order to elliminate one of the cones, eg. the one extending down the negative z axis, you need to add a condition to the line test function so that it only returns solutions which corresonds to a point on the positive z-axis.

If a given solution α_1 or α_2 gives a Z-value on the negative z-axis then discard that solution.

For the second function, getNormal, you need to analytically derive an expression for the gradient of S for any given point $\vec{p}$. Ie, the vector consisting of the first order derivatives with respect to x,y,z respectively.

$$\begin{pmatrix} \frac{df(\vec{p})}{dx} \\ \frac{df(\vec{p})}{dy} \\ \frac{df(\vec{p})}{dz} \end{pmatrix}$$

The third function, isInside, is perhaps the simplest function and is used for CSG (Constructive Solid Geometry) objects. A point $\vec{p}$ is inside if and only if $f(\vec{p}) \leq 0$. Remember to also check that a point is on the positive z-axis as a requirement to be inside the cone.

Note that you can look inside the sphere.cc class for an examples on how these functions are done on spheres – which are, surprisingly enough, almost the same as a cones!

Exercise 3.2.2

Using the infinitly large cone in your previous exercise and some CSG (Constructive Solid Geometry) it is easy to create a cone of finite height. Do this by taking the intersection of your cone and a plane with a normal along the positive z-axis and a suitable offset to create a cone of height 1. Use a transformation object to rotate and place this cone with it's apex touching the ground. It should be possible to see all sides as well as the base of the cone by moving the camera with the mouse.

Another way of doing this is by adding an intersection with a plane already in the cone object – but that would be an entirely different exercise.

Bonus exercise - implement transparancy

If you would like to, you may do this extra exercise. You won't get any rewards or cake for it, but it might be fun.

Add two new properties, *transparency* and *index_of_refraction*, to the lighting properties used by all materials and the basic raytracer. Let the raytracer use the returned transparency value to determine if it should cast another transparency ray through the object and add the resulting colour to the current pixel value.

Do this with a similar check as to how it is determined if a reflection ray is cast (look at the accumulated value with which a ray contributes to the final pixel value) and cast a ray if the transparency is high enough.

The direction in which a transparency ray should be cast can be determined using the materials index of refraction as well as the *current* index of refraction (this value needs to be propagated

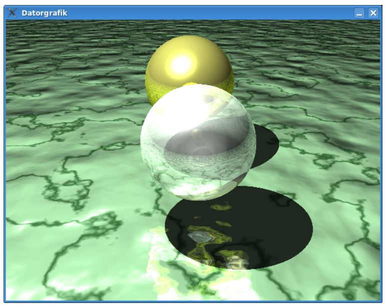


Figure 4: Added transparency and bumpmaps

through all calls to the raytracing function). Assume that the starting current index of refraction is 1.0 (air). To determine the direction in which a transparency ray is cast use Snell's law:

$$sin(\theta_1)/sin(\theta_2) = \eta_1/\eta_2$$

where θ_1 is the angle between the normal and the incoming vector and θ_2 is the angle between the normal and the outgoing vector.

Let $\vec{N}$ be the normal of the surface, $\vec{V_1}$ the incoming vector, η_1 incoming index of refraction,$\vec{V_2}$ the outgoing vector, η_2 the outgoing index of refraction. We then have:

$$\cos\theta_1 = \vec{N} \cdot (-V_1)$$

$$\cos\theta_2 = \sqrt{1 - (\frac{\eta_1}{\eta_2})^2(1 - (\cos\theta_1)^2)}$$

$$\vec{V_2} = (\frac{\eta_1}{\eta_2})\vec{V_1} + \left(\frac{\eta_1}{\eta_2}\cos\theta_1 - \cos\theta_2\right)\vec{N}$$

when $\vec{N} \cdot (-\vec{V_1})$ is positive, otherwise:

$$\vec{V_2} = (\frac{\eta_1}{\eta_2})\vec{V_1} - \left(\frac{\eta_1}{\eta_2}\cos\theta_1 + \cos\theta_2\right)\vec{N}$$

Note that the later case can occur when the transmission ray goes from the inside of an object to the outside (the gradient function normally give a normal always pointing "out" from an object), so you should check the direction $\vec{N} \cdot (-\vec{V_1})$ to know which expression to use for $\vec{V_2}$ and the two indexes of refraction.

Obviously, when $\cos\theta_2$ lacks solutions or is outside the range $[-1, 1]$ then the transmission ray is not refracted but caught by the surface of the object.

You should also handle transparency when computing shadows, in this case the effect of transparency only becomes a colouring of the light. Multiply the light colour with the transparent colour of all opaque object in a *straight line* from the point of interest to the light source. Note that it is impossible here to compute refracted shadows without resolving to photon maps or other advanced techniques.

Note that although the mathematics for this exercise may look intimidating, the amount of code needed to implement it is only about one page.

Examination

To pass the lab you must hand in a demonstrate and hand in a report covering exercise 3.1 and 3.2.2. Both the report and the demonstration must pass in order for you to pass the lab.
Recommended time: One week
Latest examination date: 2013-01-16

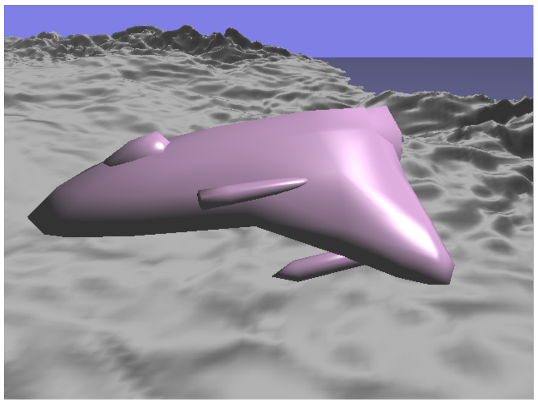


Figure 5: A small and ugly flight simulator

Lab 4 - PRELIMINARY!

Geometry shaders, Landscapes and Simulators

In this lab you will use what you have learned in the earlier labs to make a simple flight simulator. You will start from a code skeleton which includes functionality to (fractally) generate a terrain which is displayed as a mesh grid. It also includes a simple physics model for an airplane and controls for flying[4] over the landscape. Your task here is to extend the flight simulator to draw a more realistic terrain and add whatever else is needed for a simulator or game.

Start by downloading and testing `lab-4.zip` from the course web page. When you first start the program you will see a *very* poorly rendered plane flying a bit over the ground. You can steer with the numerical keyboard. You can use plus/minus to increase/decrease the motor effect and use the spacebar to switch between first/third person view. In third the later you can click and drag the mouse to rotate the camera around the plane and use the mousewheel to move the camera closer/further away from the plane.

In the code you will find 5 different shader files. For the model objects (eg. the plane in thirdpersonview) there are the normal vertex and fragement shaders. For the landscape that you will implement there are in addition to the vertex and fragment shader now also a third shader, the geometry shader.

The geometry shader is a GLSL kernel that is executed once for each graphic primitive that is triggered from the CPU and that create new *graphic primitives*. In the GLSL pipeline the geometry shader is executed *after* the vertex shader, and thus the vertex shader can pre-process the attributes passed from the CPU before they are consumed by the geometry shader.

In this lab we use the CPU to trigger the drawing of a large number of point objects, each with a unique identifier but no (other) vertex attributes of interest. These point objects correspond to `gridwidth * gridheight` cells in the world that are to be drawn. This is done for you in the lab skeleton by using the `glDrawArraysInstanced` operation which draws

[4]The parameters in the simulator and the default plane is in dire need of fine tuning, or possibly a re-implementation of the basic physics model.

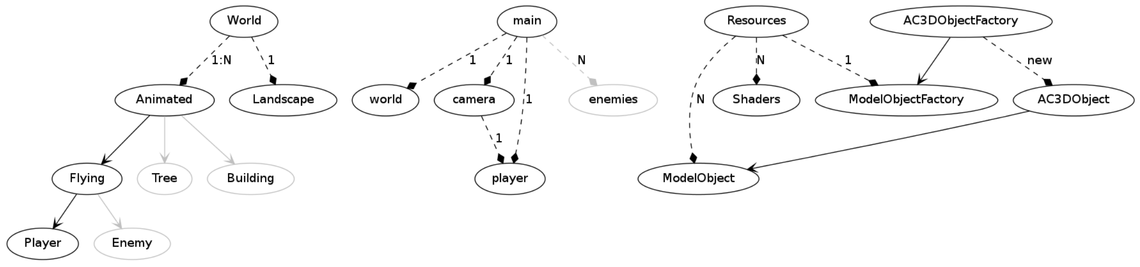


Figure 6: Classes and objects of interest in the flight simulator. In gray suggestions for future classes beyond the scope of this lab.

the same array (containing just one point) a large number of times – and passes an identifier `gl_InstanceID` to the vertex shader with this ID. The vertex shader passes this instance ID to the geometry shader.

From the integer ID each geometry shader invokation can compute which world X,Y cell that it should operate on, and compute[5] giving the graphic primitives that are to be drawn in that cell.

Note that our geometry shader is invoked with points as input argument, but can generate line_strips or triangle_strips as outputs

In order for the geometry shader and the flight simulator to agree on the appearance of the world they must both have access to a *heightmap* that determines the height of each part of the world. This height map is created by the `Landscape` class and is passed as a texture into GLSL. The geometry shader can use this texture to generate correct vertex attributes such as positions and normals.

The class `Player` is used to represent the user and his view of the airplane. The class `Landscape` represents the maps and handles the visualization, it is primarily in `Landscape::draw` and the different shaders that you will be working with your changes.

Note before starting: if the initial version of the code is *way* to slow in your computer, you can change the width and height specified in `landscape.cc:40`. Use eg. 512 or 256 to make the code 4 respectively 16 times faster. Also note that the program takes a few seconds in the begining to generate the landscape, smaller landscape sizes will make this faster.

Exercise 4.1 Filling in the landscape.
Start by looking at `landscapeGeometryShader.glsl`. This shader currently generates the lines the build up the mesh of the landscape. Your first exercis will be to modify this shader and the corresponding `landscapeFragmentShader.glsl` to make a filled in landscape with proper lights. To do (just like before). this, you need the following changes:

1. Change the `layout` qualifier of the geometry shader to generate `triangle_strip` primitives and change the number of maximum allowed out vertices of the shader.

2. Change the `main` function to give the "correct" vertices for the triangle strips that are

[5]Since we draw `W*H` primitives, we can compute `y=id/W, x=id%W`

drawn.

3. Verify that it works, that you get a completely filled in landscape with a uniform colour.

4. Create new `out` variable that give per vertex attributes to the fragment shader. You will want to pass out atleast the world coordinate of the points and their normals.

5. Implement phong shading in the fragment shader like in your previous exercises.

6. Finally, re-use your old code for drawing model objects efficiently through the use of a vertex buffer object and let the `Flying::draw` routine draw a fighter when the simulation is in third person view. In order for the fighter to be rotated correctly when it moves you need to create a rotation matrix. Use the `forward, right, up` vectors as the X,Y and Z columns of this vector. Note that the 3D model currently does not have the Y axis pointing forwards which the simulator expects - so you need to rotate it to face forward first before applying the other rotations.

Additionally, you may want to change the landscape to have a simple "water" appearance at $z = 0.0$. You can do this by testing for the z-coordinate in the geometry shader and change it to zero, pass in a correct water normal and in the fragment shader test use different light properties for the water or the landscape.

If you have done this exercise properly you should now have a simple flight simulator, where you can fly in first person or third person view over a landscape. However, it is probably not very fast (depending on your computer).

Note. This exercise does not need to be included in the report or demonstrated.

Exercise 4.2 Generating more triangles, faster.
In this exercise you will make some optimisations that will make your landscape drawing routines much faster. On an ATI 6680 card it will mean the change between 19fps and 80fps.

Begin by modifying your geometry shader to work on *groups* of patches that are to be drawn. To do this you should first modify the landscape drawing code to scale down the width and height of the grid that is to be drawn by a factor of 4 [6]. Next, modify the computation of the X and Y position that this patch corresponds to in the geometry shader by multiplying with the same factor. You geometry shaders will now be invoked 4*4 times less than before, so we need to modify it to draw 4*4 rectangles on each invokation.

Continue by modifying the `main` function in your geometry shader to draw *four different geometry strips* (separated by a call to `EndPrimitive`) corresponding to an offset of 0,1,2,3 from the given Y value. Inside each triangle strip, it should now step over the five (!) X offset values 0,1,2,3,4 – resulting in four pairs of triangles (one per rectangle) covering the full area that needs to be drawn by this geometry shader.

Note that you must modify the `max_vertices` argument in your geometry shader, otherwise you will not see all vertices.

[6]Depending on your GPU, it may be more efficient with a factor of 2,4,8 or 16 – the limitation has to do with the maximum number of allowed vertices that a single geometry shader may generate

If you have done this exercise properly you should now have a faster flight simulator, but that otherwise gives *exactly* the same result as from your previous exercise.

Note. This exercise does not need to be included in the report or demonstrated.

Exercise 4.3 Frustum culling.
In this exercise you will make some more optimisations that will make your landscape drawing routines even faster by *not generating triangels that are outside the screen*. On an ATI 6680 card it will mean the change between 80fps and 200fps.

Start by modifying your geometry shader to compute the screen coordinates of the four vertices that corresponds to the corners of the whole patch that is to be drawn (ie. the whole set of 4x4 triangles). Note that you need to manually perform the normalization of the homogeneous coordinate.

Compute the *axis aligned clipping bounding box* of these coordinates by taking the `min` and `max` values in x,y,z. Finally, check if any of the smallest x,y,z coordinates are greater than +1.0, or if any of the biggest x,y,z coordinates are smaller than -1.0. Iff this check holds true then the bounding box is outside the screen and you can discard generating any triangles.

If you have implemented this exercise correctly your code should go approximately 2-3 times as fast. You may get some artifacts close to the edges of the screen, to avoid these artifacts you can use a slightly higher value than the 1.0 above.

This exercise DOES need to be included in the report or demonstrated.

Exercise 4.4 Procedural textures.
In this exercise you will implement a simple for of a *procedural texture* in the landscape fragment shader. In the lab skeleton you have been given a *noise* function that gives pseudo-random values for any integer valued points in a 3D space, and with smoothly interpolated values for the fractional points in between.

Start by defining a floating point value `hasGrass` that should be one on any flat surfaces and zero on any (completely) vertical surface. You can do this in the fragment shader by relying on the normal that you have been passed in.

Next, define your ambient and diffuse colour properties to be linearly dependent on this variable, interpolating between two constant expressions (vec4) that correspond to grass respectively dirt or stone. (You have to come up with a few colour values yourself here).

Hint: The built in function `mix` is quite useful for this, and it can work on vectors of data.

Next, continue by adding a level of noise to your `hasGrass` variable. The noise should cover a number of octaves where each octave has twice the frequency of the previous octave and half the amplitude of the previous octave. Let the starting octave have a frequency of roughly 1 change per world unit, and an amplitude of 0.5.

If you want to (not required), you can change your geometry shader to pass in a “low pass” normal, that uses a larger offset in the textures when approximating the derivative. This larger offset can make the texture better. Also, (if you want to) you can incorporate the height of a fragment to make the mountain tops covered in snow.

This exercise DOES need to be included in the report or demonstrated.

Examination

Recommended time: 2-3 weeks
Latest examination date: 2013-01-16