IN4152 - 3D Game Project

Introduction

In your final project the goal is to create a three-dimensional game. In this game you will use all of the past skills you have acquired from the previous practicals. Above are some examples of a game you can take inspiration from, but feel free to do something completely different as long as you make some implementation of the requirements outlined below:

Requirements

Your game should feature a **main character, modeled by you [Exercise 2]**, that can move around in 3D space using some input mechanism like the keyboard or mouse. The **rotation and translation** of the player should be accomplished using **matrix transformations [Exercise 1]**. The view the player has on the scene is determined by setting **projection and view matrices [Exercise 1].** You might want a first-person or third-person or birds-eye type of game, therefore the position of the camera is up to you.

A substantial portion of the environment in which the game takes place should be **procedurally generated**. For example, this means you should **generate a terrain [Exercise 4]** out of triangles and make it vary with some math functions like you have done in the practical. The game should start at the **house that you built in the Blender practical exercise [Exercise 2].**

The environment and objects inhabiting it should be shaded in ways you have been taught during the course. This means you should implement the **(Blinn-)Phong Shading Model [Exercise 3]** with a diffuse and specular term. In addition at least one character should be shaded using the **Toon-Shading model [Exercise 3&5]**. In addition to the shading, some of the objects in the scene should be **textured [Exercise 4]**. You can download the textures from the internet or make them yourself. The scene would look rather fake with just shading as objects don't cast shadows on each other. Therefore, you should simulate sunlight by applying **shadow mapping [Exercise 6]**. The shadows will probably look pretty rough, so add some kind of **filtering** (like PCF) **[Exercise 6]** to them to make them look better.

The game should contain at least one other character (possibly an end-boss) that consists of **several animated components [Exercise 1]** (e.g., a robot arm or a snake, or many objects rotating like a solar system). The 3D models required for these other characters (and potentially your own character) can be downloaded from the internet.

For a bonus, add other cool graphic features, such as (but not limited to):

- Export your animated model by dumping individual frames to .obj files **[Exercise 2]**
- Particle effects (explosions, magic spells, fire)
- Point-lights (like torches or light bulbs)
- The illusion of infinite terrain
- Post-processing effects
- More complex shading techniques, such as normal mapping
- Terrain based on a 3D height field
- Animated textures (by switching textures frame to frame)
- Zoom effects / perspective changes

- Changing lighting conditions (like a day-night system)
- A minimap (by rendering from a camera high up in the sky, and displaying it on a quad)
- An interactive user-interface
- Collision between the player and objects
- Generate a maze or dungeon for the player to move through
- Generate environment props like buildings procedurally.

Feel free to come up with your own extra effects and features, the above list is by far not exhaustive of all the possibilities in computer graphics and game development!

For the project you are not judged on the length, fun, functionality, or quality of the gameplay, but rather on the graphics techniques you used and how well they were applied. In short, a boring game can receive a good grade! Of course, a nice presentation, creativity, or breathtaking visuals are a bonus.

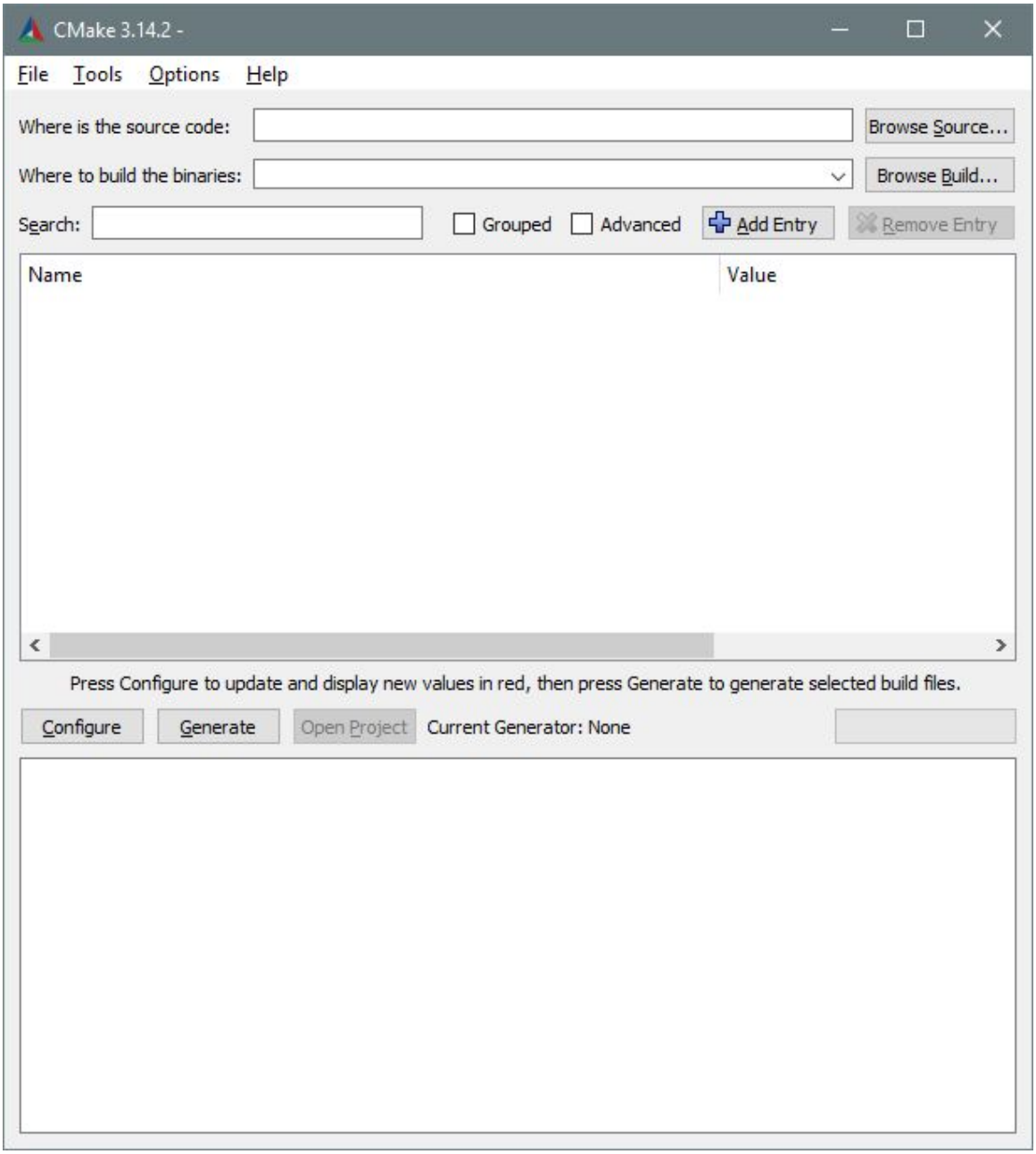
It is advised to work in **groups of two-three people**, three is the recommended number (larger groups receive a penalty).
Please upload your contribution before the start of the last session, which will be scheduled on the 14th of June. On this date we will also schedule **a short multiple choice exam of 30 minutes, registration via Osiris is not needed**. The project upload should include: the **source code**, a **short report** (max 200 words) of the **implemented techniques each accompanied by one or multiple figures** and including a **list of the work done by each individual group member** (not included in the word count).

Please also prepare a **5 minute presentation** (including a **live demo**) of your work to present during the last practical session.

Framework

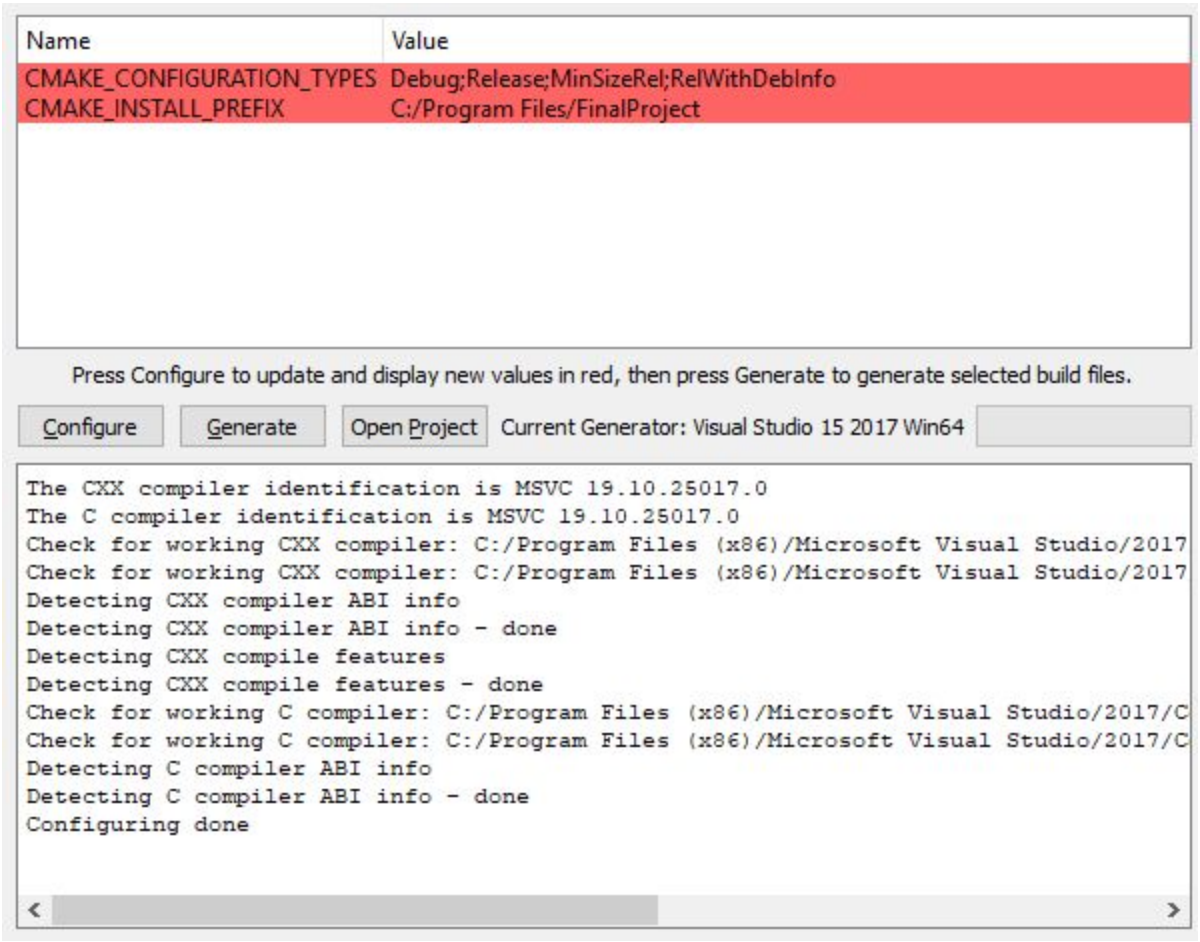
For the creation of your game you are allowed to use any of the past frameworks of the practical assignments. However, we also provide a framework specifically for this exercise which only allows **Modern OpenGL (version 3.3)** code, but gives you several useful classes to get going more easily.

The framework can be built by downloading and installing CMake (https://cmake.org/download/).



When starting CMake you will see the window above. In order to build the framework with this follow these steps:

1. Click **Browse Source** and point it to the folder containing your final project files (Source, Resources, CMakeLists.txt, etc.)
2. Make a folder called **Build** in that directory and point the second field to it by pressing **Browse Build**.
3. Now press the **Configure** button
4. In the generator dropdown, select **Visual Studio 15 2017 Win64** on Windows or **XCode** on Mac OS and then press **Finish**.
5. If all goes well your window should now look similar to this:



6. Now press **Generate** and **Open Project** which should open Visual Studio together with the framework.
7. Lastly, right-click **FinalProject** in the **Solution Explorer** and **Set as StartUp Project**
8. Optionally, you can set your build from **Debug** to **Release** at the top, if you want speed over debugging capabilities.

Helper classes

Vector3f

A Vector containing 3 floating-point components. Supports the following operations:

- **Vector3f(x, y, z)** - Creates a vector with the given x,y,z components.
- **Vector3f(xyz)** - Creates a vector with all components set to xyz.
- **.length()** - Computes the euclidean length of the vector.
- **dot(v1, v2)** - Calculates the dot product of two vectors.
- **cross(v1, v2)** - Calculates the cross product of two vectors.
- **normalize(v)** - Normalizes the given vector.

Furthermore, it supports addition, subtraction, multiplication and division with **Vector3f** and **float** using the normal operators.

Matrix4f

A 4x4 Matrix containing float-point elements. Supports the following operations:

- **Matrix()** - Creates an identity matrix.
- **.setIdentity()** - Explicitly set the matrix to the identity matrix.
- **.translate(v)** - Build a translation matrix with the given Vector3f as the translation.
- **.rotate(angle, x, y, z)** - Build a rotation matrix for rotation of ***angle*** around ***axis*** (x,y,z).
- **.scale(scale)** - Build a scaling matrix for uniform scaling by the given float value.
- **.transform(v, w)** - Transform the Vector3f ***v*** by the matrix, given a w-component of 0 or 1.
- **transpose(m)** - Calculates the transpose of a given matrix.
- **determinant(m)** **-** Calculates the determinant of a given matrix m
- **inverse(m)** - Calculates the inverse of a given matrix m

Furthermore, the matrix supports multiplication with other matrices using the ***** operator, and each element can be individually changed using the **[i]** operator. Here ***i*** is a value between [0, 15] and is the **i'th** element of a column-major 4x4 matrix that looks like this:

[0 4 8 12]
[1 5 9 13]
[2 6 10 14]
[3 7 11 15]

ShaderProgram

A shader that abstracts away all the OpenGL functions related to shader management. New shaders can be added by following the code already present in the project:

```
try {
    shaderProgram.create();
    shaderProgram.addShader(VERTEX, "Resources/shader.vert");
    shaderProgram.addShader(FRAGMENT, "Resources/shader.frag");
    shaderProgram.build();

    shadowProgram.create();
    shadowProgram.addShader(VERTEX, "Resources/shadow.vert");
    shadowProgram.addShader(FRAGMENT, "Resources/shadow.frag");
    shadowProgram.build();

    // Any new shaders can be added below in similar fashion
    // ....
}
catch (ShaderLoadingException e)
{
    std::cerr << e.what() << std::endl;
}
```

The shader supports uploading many kinds of **uniform** values to the shader. These are some typical functions:

- **.uniform3f("name", v)** - Uploads the given Vector3f ***v*** to the uniform ***vec3*** variable in your shader file called ***name***.
- **.uniform3f("name", x, y, z)** - Uploads the given ***x, y, z*** values to the uniform ***vec3*** variable in your shader file called ***name***.
- **.uniformMatrix4f("name", m)** - Uploads the given Matrix4f **m** to the uniform ***mat4*** variable in your shader file called ***name***.

In order to render objects with your shader of choice, or to upload uniforms to it, the shader must be bound. This is performed by calling:

- **.bind()** - Binds the shader, any consecutive draw calls are drawn with this shader.
- **.release()** - Unbinds the shader, further draw calls are invalid until another shader is bound.

Calling **bind()** on a shader automatically unbinds any other shaders that were bound, therefore in most cases it is not necessary to call **release()**.

Images and Models

Images and models can be loaded using the provided functions inside **Image.h** and **Model.h.**

- **loadImage(“pathToImage”)** - Returns an Image object containing a **width** and **height** and a **handle** to an OpenGL texture object.
- **loadModel(“pathToModel”)** - Returns a Model object containing **vertices**, **normals**, and **texCoords** and a **handle** to an OpenGL vertex array object, which you need for drawing.