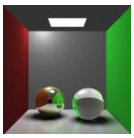


2024/2025, 4th period

INFOGR: Graphics

Practical 2: Rasterization

Author: Peter Vangorp, based on a previous version by Jacco Bikker

The assignment:

The purpose of this assignment is to create a small OpenGL-based 3D engine, starting with the provided template. The renderer should be able to visualize a scene graph, with (potentially) a unique texture and shader per scene graph node. The shaders should at least support the full Phong illumination model. For a full list of required functionality, see section "Minimum Requirements".

As with the first assignment, the following rules for submission apply:

- Your code has to compile and run on other machines than just your own. If this requirement isn't met, we may not be able to grade your work, in which case your grade will default to 0. Common reasons for this to fail are hardcoded paths to files on your machine.
- Please **clean** your solution before submitting (i.e. remove all the compiled files and intermediate output). This can easily be achieved by running `clean.bat` (included with the template). After this you can zip the solution directories and submit them on Blackboard. If your zip-file is multiple megabytes in size you've included large assets or something went wrong (not cleaned properly).
- We want to see a consistent and readable coding style: formatting; descriptive names for variables, methods, and classes; and comments. Most code editors have tools to help with formatting and indentation, and with renaming things ("refactoring") if necessary.

> *"Programs are meant to be read by humans*
> *and only incidentally for computers to execute."*
> *– Structure and Interpretation of Computer Programs*

Grading:

If you implement the minimum requirements, and stick to the above rules, you score a 6.

We deduct points for: a missing readme.txt file, a solution that was not cleaned, a solution that does not compile, a solution that crashes, inconsistent coding style, insufficient comments to explain the code, or incorrectly implemented features.

Implement additional features to earn additional points (up to a 10).

Deliverables:

A ZIP-file containing:

1. **The contents of your (cleaned) solution directory**
2. **The readme.txt file**

The contents of the solution directory should contain:

(a) Your **solution file** (.sln)

(b) All your **source code**

(c) All your **project** and **content files** (including shaders, models, and textures)**.**

The readme file should contain:

(a) The names and student IDs of your team members.

[1–3 students – the team does not have to be the same as for P1]

(b) A statement about what minimum requirements and bonus assignments you have implemented (if any) and information that is needed to grade them, including detailed information on your implementation.

[We will not search for features in your code. If we can't find and understand them easily, they may not be graded, so make sure your description and/or comments are clear.]

(c) A list of materials you used to implement the 3D engine. If you borrowed code or ideas from websites or books, make sure you provide a full and accurate overview of this.
Considering the large number of OpenGL rasterizers available on the internet, we will carefully check for original work.

Put the solution directories and the readme.txt file directly in the **root** of the zip file.

Teamwork: If you use the Git version control system for teamwork, for example on GitHub or GitLab, be aware that by default it doesn't put .obj files in the repository because it assumes .obj files are intermediate outputs generated by the compiler. But the template uses the .obj file format for meshes in the assets folder. Edit your .gitignore file if you want to put those assets in the repository.

Mode of submission:

- Upload your zip file before the deadline via Blackboard. The Blackboard software allows you to upload without submitting: please do not forget to hit 'submit' once you are sure we should see the final result. Please do not forget the final submit!
- Re-download your submission from Blackboard, unzip it into a different folder, and check that it runs and looks like the version you intended to submit. This catches most mistakes like missing files or submitting the wrong version. You can correct these mistakes and re-submit until the deadline.

Note that we only grade the last submitted version of your assignment.

Deadline:

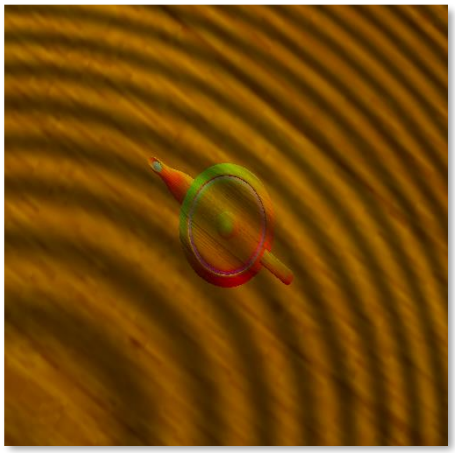
Friday, June 27, 2025, 17:00h

This is a hard deadline. If you miss this deadline, your work will not be graded.

Time management: Don't postpone working on this assignment. It only increases the pressure and stress, and you may run out of time.

Fraud & plagiarism:

- Never look at other students' code. Don't discuss implementation in detail. Reference every source in your readme.txt and/or in code comments.
- We use automated content similarity detection tools to compare all submissions of this year and previous years, and online sources. All suspected cases of fraud or plagiarism must be reported to the Board of Examiners.



High-level Outline

For this assignment you will implement a basic OpenGL-based 3D engine. The 3D engine is a tool to visualize a scene graph: a hierarchy of meshes, each of which can have a unique local transform. Each mesh will have a texture and a shader. The input for the shader includes a set of light sources. The shading model implemented in the fragment shader determines the response of the materials to these lights.

The main concepts you will apply in this assignment are *matrix transforms* and *shading models*.

Matrix transforms: objects are defined in *local space* (also known as *object space*). An object can have an orientation and position relative to its *parent* in the *scene graph*. This way, the wheel of a car can spin, while it moves with the car. In the real world, many moving objects move relative to other objects, which may also move relative to some other object. In a 3D engine, we have an extra complication: after we transform our vertices to *world space*, we need to transform them to *camera space*, and then to *screen space* for final display. A correct implementation and full understanding of this pipeline is an important aspect of both theory and practice in the second half of the course.

Shading: using *interpolated normals* and a set of *point lights* we can get fairly realistic materials by applying the *Phong lighting model*. This model combines *ambient lighting*, *diffuse reflection* and *glossy reflection*. Optionally, this can be combined with *texturing* and *normal mapping* for detailed surfaces. A good understanding of concepts from ray tracing will also be useful here.

The remainder of this document describes the C# template, the minimum requirements for the assignment and bonus challenges.

Finally, you may work on **post processing.** In the screenshot at the top of the page you see the effect of a dummy post processing shader, which you can find in shaders/fs_post.glsl. Its main functionality is the following line:

```
outputColor *= sin( dist * 50 ) * 0.25f + 0.75f;
```

Disable this line to get rid of the ripples. Replace it by something more interesting to get extra points: see the last page of this document for details.

You're **not** expected to re-implement any features that are already provided by OpenTK, GLSL, or the template. Such features include basic vector and matrix math, and loading texture images and meshes.

Template

For this assignment, a fresh template has been prepared for you.

When you start the template, you will notice that quite some work has been done for you:

- Two 3D models are loaded. The models are stored in the text-based OBJ file format, which stores vertex positions, vertex normals and texture coordinates.
- A mesh class is provided that stores this data for individual meshes.
- A texture and shader class is also provided.
- Dummy shaders are provided that use all data: the texture, vertex normals, and vertex coordinates.

In short, the whole data pipeline is in place, and you can focus on the functionality for this assignment. Let's have a closer look at the provided functionality:

class Texture: this class loads common image file formats (.png, .jpg, .bmp etc.) and converts them to an OpenGL texture. Like all resources in OpenGL, a texture simply gets an integer identifier, which is stored in the public member variable 'id'.

class Shader: this class encapsulates the shader loading and compilation functionality. It is programmed to work with the included shaders: e.g., it expects certain variables to exist in the shader, such as per-vertex data (position, normal, texture coordinates), and "uniform" transformation matrices. You may need to add more variables in the shaders and correspondingly in this class.

class Mesh: this class contains the functionality to render a mesh. This includes Vertex Buffer Object (VBO) creation and all the function calls needed to feed this data to the GPU. The render method takes a shader, transformation matrices, and a texture, which is all you need to draw the mesh. Note that this means that each mesh can use only a single texture. Currently the Assimp library is used to load simple meshes from supported file formats. Feel free to improve the mesh loading code.

class MyApplication: you will find some ready-made functionality here. To demonstrate how to use the other classes, a texture, a shader and two meshes are loaded and displayed with a dummy transform. This definitely needs some work (just like the dummy shaders).

The template will produce informative OpenGL error messages to help with debugging.

Parts of the template are implemented in two functionally equivalent versions. You can choose which version should be used by setting the constant `OpenTKApp.allowPrehistoricOpenGL`:

- `true`: Use deprecated code that is usually shorter and easier to understand but that is not supported anymore on Apple devices and should only be used in legacy codebases. If you choose this version, also *your* code may use deprecated code ("Compatibility profile").
- `false`: Use Modern OpenGL code that is usually longer and more difficult to understand but that is supported everywhere and should be used in new codebases. If you choose this version, also *your* code must use Modern OpenGL ("Core profile").

Your Task

You should first familiarize yourself with all the provided functionality and code.

You then have two main tasks for this assignment:

1. Implement a scene graph;
2. Implement a proper shader;

Demonstrate all the functionality you add with a demo scene and screenshots.

Scene graph: currently, the application renders two objects, but this is entirely hardcoded. Your task is to add a new *class SceneGraph*, which stores a hierarchy of meshes. The mesh class needs to be expanded a bit as well; each mesh should have a local transform. The SceneGraph class should implement a Render method, which takes a camera matrix as input. This method then renders all meshes in the hierarchy. To determine the final transform for each mesh, matrix concatenation should be used to combine all matrices, starting with the camera matrix, all the way down to each individual mesh.

Task list for the scene graph:

1. Add a model matrix to the Mesh class.
2. Add the Scene Graph: a data structure for storing a tree-structured hierarchy of meshes, where the position of each mesh in the scene will also be affected by the model matrices of all its ancestors.
3. Add a Render method for the Scene Graph that recursively processes the nodes in the tree, while combining matrices so that each mesh is drawn using the correct combined matrix.
4. Call the Render method of the Scene Graph from the Game class, using a camera matrix that is updated based on user input.

Shader: the dummy shaders combine the texture with the normal. As you may have noticed, the normal is directly converted to an RGB color (a useful debug visualization to inspect the 3 component values of the normal vector, but of course this is not a realistic material). Your task is to replace this dummy shader with a full implementation of the Phong lighting model. This means that you need to combine an ambient color with the summed contribution of one or more light sources.

Task list for the shader:

1. Add a uniform variable to the fragment shader to pass the ambient light color.
2. Add a Light class. Perhaps it would be nice if lights could also be in the scene graph.
3. Either add a hardcoded static light source to the shader, or (for extra points) add uniform variables to the fragment shader to pass light positions and colors. Don't over-engineer this; if your shader can handle 4 lights using four sets of uniform variables, you meet the requirements to obtain the bonus points.
4. Implement the Phong lighting model.

Demonstration: once the basic 3D engine is complete, it is time to showcase its capabilities. Build a small demo that shows the scene graph and shader functionality.

Minimum Requirements

To pass this assignment, we need to see:

Camera:

- The camera must be interactive with keyboard and/or mouse control. It must at least support translation and rotation.

Scene graph:

- Your demo must show a hierarchy of objects. The scene graph must be able to hold any number of meshes, and may not put any restrictions on the maximum depth of the scene graph.

Shaders:

- You must provide at least one correct shader that implements the Phong shading model. This includes ambient light, diffuse reflection and glossy reflection of the point lights in the scene. To pass, you may use a single hardcoded light.

Demonstration scene:

- All engine functionality you implement must be visible in the demo. A high quality demo will increase your grade.

Documentation:

- Describe which features you implemented. Describe the controls for your demo.

Bonus Assignments

Meeting the minimum requirements earns you a 6 (assuming practical details are all in order). An additional four points can be earned by implementing bonus features. An incomplete list of options, with an indication of the difficulty level:

- [EASY] Multiple lights (at least 4), which can be modified at run-time (0.5 pt)
- [EASY] Spotlights (0.5 pt)
- [EASY] Environment mapping to show a cube map or sphere map texture in the background and/or in mirror reflections (0.5 pt)
  NOTE: The mapping from screen pixel coordinates to environment map texture coordinates must be computed in the shader. You should not use a simple textured cube or sphere mesh.
- [MEDIUM] Frustum culling in the scene graph render method (1 pt)
- [MEDIUM] Normal mapping (1 pt)
- [HARD] Shadows using shadow mapping (1.5 pt)

Additional challenges related to post processing:

- [EASY] Vignetting and chromatic aberration (0.5 pt)
- [MEDIUM] Generic color grading using a color look-up table (1 pt)
- [MEDIUM] A separable blur filter with variable kernel width (1 pt)
- [MEDIUM] HDR glow (requires blur filter and HDR render target) (1pt)
- [HARD] Depth of field (requires blur filter) (1.5 pt)
- [HARD] Ambient occlusion (1.5 pt)

Important: many of these features require that you investigate these yourself, i.e., they are not necessarily covered in the lectures or in the book. You may of course discuss these on Teams to get some help. Most websites, tutorial videos, and AI chatbots give code that doesn't meet some of the specified requirements.

Obviously, there are many other things that could be implemented in a 3D engine. Make sure you clearly describe functionality in your report, and if you want to be sure, consult the lecturer for reward details.

Do's and don'ts

DO	DON'T
Use the Vector{2,3,4} and Matrix4 types from OpenTK.Mathematics	Don't implement your own Vector{2,3,4} or Matrix4 classes, because it may be buggy or slow, and it won't give you a higher grade anyway
Use the vector/matrix math operations provided by the Vector{2,3,4} and Matrix4 types	Don't implement your own math operations using the separate values stored in those types, because the Matrix4 type might use "the other convention", your implementation may be buggy or slow, and it won't give you a higher grade anyway
Normalize every vector, unless • **you've checked that the math works for unnormalized vectors, or** **you're certain that it's already normalized**	Don't assume that the math works for unnormalized vectors Don't assume that vector parameters passed to a function or shader have already been normalized
Use a consistent naming convention that explicitly states the space for all points/vectors/normals and the from- and to-spaces for all matrices • **Concatenate correctly by playing dominos**	Mix naming conventions or omit that information, because you will make mistakes and because we can't give partial credit if we can't understand what you were trying to do
Use single-precision floating point, meaning float, Vector{2,3,4}, and Matrix4 types and MathF functions	Don't use double (e.g., floating point constants without the suffix f), the Vector3d type, or Math (without the suffix F) functions, because they tend to be slower and that much precision isn't needed in rendering GLSL only has single precision float and vec2/vec3/vec4 types and doesn't allow constants with a suffix (f)
Include the readme.txt file inside your ZIP file	Don't include the readme.txt as a comment or as a separate file on Blackboard, because it might be lost
Evaluate critically whether the result looks correct and comment in the readme.txt if a feature doesn't quite look right	Don't just tick features if they don't quite look right, because we will notice that but we can give partial credit if you noticed too
Submit your whole project folder, including source code, shaders, assets, and IDE-related files	Don't omit any files, e.g., the template files or the shaders, because we should be able to run your project as submitted
Include the readme.txt, clean the project, write comments where necessary, use formatting and clear naming to make the code more readable, etc.	Don't miss out on the easy points that you could earn for making our grading work a bit easier

And Finally…

Don't forget to have fun; make something beautiful!