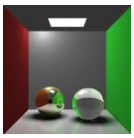


2021/2022, 4th period

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 multiple lights.

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 send them over. If your zip-file is multiple megabytes in size something went wrong (not cleaned properly).
- When grading, we want to get the impression that you really understand what is happening in your code, so your source files should also contain comments to explain what you think is happening.
- Finally, we also want to see a consistent and readable coding style. Use indentation to indicate structure in the code for example.

Grading:

If you do all the above properly, you get a 6. Implement additional features to obtain additional points (up to a 10).

Additional grading details:

- From the base grade of 6, we deduct points for a missing readme.txt file, a solution that was not cleaned, a solution that does not compile, or a solution that crashes (1 point for each problem).
- Up to 1 point may be deducted for an inconsistent coding style.
- Up to 1 point may be deducted for code that is not properly commented.

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 or 2 students. The team does not have to be the same as for P1, feel free to shuffle]

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

[If we can't find features you implemented (and understand them easily), they will 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, make sure you provide a full and accurate overview of this.

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

Mode of submission:

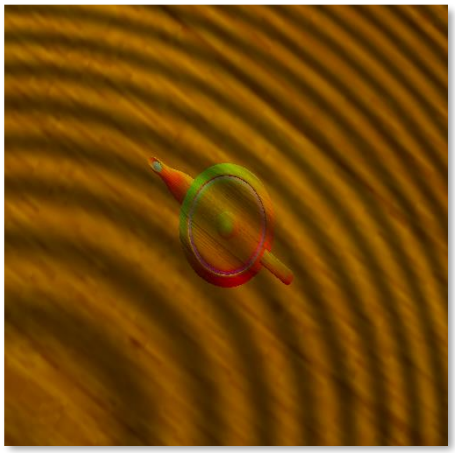
- Upload your zip file to Blackboard and make sure to click Submit.
- If you send an updated version, we only grade the last submitted version of your assignment. Older versions will be discarded.

Deadline:

Friday, June 24, 2022, 17:00h

If you miss this deadline, you may hand in your work up to 7 hours late, i.e., until midnight, but this decreases your grade by 0.5 points; or up to 24 hours late, i.e., until Saturday June 25, 17:00, but this decreases your grade by 1.0 point.

If you miss these deadlines as well, your work will not be graded.



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.

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 functionality.

class Texture: this class uses C# Bitmaps to load data from 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 hardwired to the included shaders: e.g., it expects certain variables to exist in the shader. Since these are exactly the variables you will need for this assignment, chances are you won't need to make any changes in this class. Expected variables are vPosition, vNormal, vUV and the 'uniform' transform. You can find these in vs.glsl, which forwards them to the fragment shader.

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, a matrix 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: in 3D engines it is common practice to split geometry in batches that use the same texture.

class MeshLoader: this is a helper class that loads OBJ files for you. Note that it is slow; the meshes that are included in the template are therefore small to reduce application startup time. Feel free to replace it with something faster if necessary.

class Game: 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).

Your Task

As mentioned in the introduction, you have two main tasks for this assignment:

1. Implement a scene graph;
2. Implement a proper shader;
3. Demonstrate the functionality.

In more detail:

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:

- Add a model matrix to the Mesh class.
- Add the SceneGraph class.
- Add a data structure for storing a hierarchy of meshes in the scene graph.
- Add a Render method to the scene graph class that recursively processes the nodes in the tree, while combining matrices so that each mesh is drawn using the correct combined matrix.
- Call the Render method of the SceneGraph 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:

- Add a uniform variable to the fragment shader to pass the ambient light color.
- Add a Light class. Perhaps it would be nice if lights could also be in the scene graph.
- 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.
- 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 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.

Demo:

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

Documentation:

- Describe your architecture: what data structures did you use, and how did you engineer the application. Describe which features you implemented. Describe the controls for your demo. Provide screenshots of the features.

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] Add multiple lights, which can be modified at run-time (0.5 pt)
- [EASY] Add spotlights (0.5 pt)
- [EASY] Add cube mapping (0.5 pt)
- [MEDIUM] Add frustum culling to the scene graph render method (1 pt)
- [MEDIUM] Add normal mapping (1 pt)
- [MEDIUM] Make the floor plane reflective using a stencil (1 pt)
- [HARD] Add shadows (1.5 pt)

Additional challenges related to post processing:

- [EASY] Add vignetting and chromatic aberration (0.5 pt)
- [MEDIUM] Add generic color grading using a color cube (1 pt)
- [MEDIUM] Add a separable box filter blur with variable kernel width (1 pt)
- [MEDIUM] Add HDR glow (requires box filter and HDR targets) (1pt)
- [HARD] Add depth of field (1.5 pt)
- [HARD] Add 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.

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.

And Finally…

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