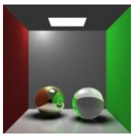


2015/2016, 4th quarter

INFOGR: Graphics

Practical 3: Rasterization

Author: Jacco Bikker

The assignment:

The purpose of this assignment is to create a small 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 and second assignment, the following rules for submission apply:

- Your code has to compile and run on the machines in the rooms allocated to the working colleges, so if you work on other computers make sure to do a quick check there before you submit it. If this requirement isn't met, your work cannot be graded and your grade will default to 0.
- Make sure you **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 mega-bytes 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. Don't worry about this too much, if it is readable and consistent throughout the whole project, you should be fine.

Grading:

If you do all the above properly, you get a 7. Implement additional features to obtain additional points (up to a 10).

Additional grading details:

- From the base grade of 7, we deduct points for a missing readme, a solution that was not cleaned, a solution that does not compile, or a solution that crashes (1 point for each problem).
- One point will be deducted if you worked alone on the assignment.
- Up to 1 point will be deducted for an inconsistent coding style.
- Up to 1 point will be deducted for code that is not properly commented.

Deliverables:

A ZIP-file containing:

1. **The contents of your (cleaned) solution directory**
2. **The read-me (in the .txt file format)**

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.

[2-3 students; penalties for submitting with less or more team members apply]

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

[We will not make any wild guesses about what you might have implemented nor will we spend long times searching for special features in your code. If we can't find 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 <u>full and accurate overview</u> of this.

Put the solution directories and the read-me file (in the .txt file format) directly in the **root** of the zip file. Note that any violation to these rules can have consequences for your grade. Also notice that the readme file should be well readable. It is part of the program that you are producing, so the rules about "consistent and clear coding style" apply to it as well.

Mode of submission:

- Upload your zip file before the deadline via the SUBMIT system at http://www.cs.uu.nl/docs/submit/
- Make sure to upload it to the correct entry, i.e. **not** the ones for late delivery if you are submitting on time (otherwise, grade deductions will still apply).
- Note that we only grade the latest submitted version of your assignment, so if you upload to the late delivery entries your earlier submission will be discarded.

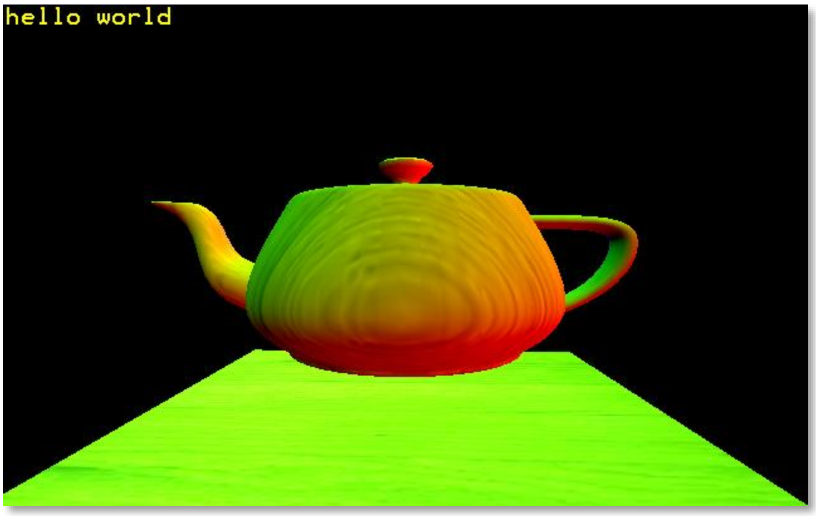
Deadline:

Tuesday, June 28, 2016, 23:59h

If you miss this deadline, there will be a second entry in the submit system to upload your solutions. It is open 12h longer, i.e. till **Wednesday, June 29, 2016, 12:00 (noon)**. Uploading to this entry will result in a deduction of 0.5 in your grading (attention: the deduction still applies if you upload to this entry earlier).

If you miss the second deadline as well, there will be a third entry in the submit system to upload your solutions. It is open 24h longer, i.e. till **Wednesday, June 29, 2016, 23:59**. Uploading to this entry will result in a deduction of 1.0 in your grading (attention: the deduction still applies if you upload to this entry earlier).

If you the miss third deadline as well, your assignment will be graded with a 0.



High-level Outline

For this assignment you will implement a basic 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 model 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 shading model. This model combines ambient lighting, diffuse reflection and glossy reflection. Optionally, this can be combined with normal mapping for detailed surfaces. We will be applying concepts from ray tracing here.

The remainder of this document describes the P3 template, the requirements for the assignment and bonus challenges.

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, encapsulating most of the functionality you built in the first assignment (P1), but with a twist: vertex data (position, normal, texture coordinates) is now interleaved, and polygons use vertex indices.
- 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 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 from P1. 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 will never have to change this class, even though it is hardly pretty or versatile. 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 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 engine land 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.

class Game: you will find some ready-made functionality here. A Stopwatch is used to make animation speed consistent. 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 view 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 (which is possible; it's a 3-component vector after all, but of course this is nonsense). Your task is to replace this dummy shader with a full implementation of the Phong shading model. This means that you need to combine an ambient color with the summed contribution of one or multiple 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 and obtain bonus points.
- Implement the Phong shading model.

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

The Full Thing

To pass this assignment, we need to see:

Camera:

- The camera must be interactive. 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.

A Bit Extra

Meeting the minimum requirements earns you a 7 (assuming practical details are all in order). An additional three points can be earned by implementing optional 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)
- [HARD] Add post-processing using an FBO (1.5 pt)

Important: many of these features require that you investigate these yourself, i.e. they are not necessarily covered in the colleges. You may of course discuss these on Slack 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!

May the Light be with you,

- *Jacco.*