NTNU
Kunnskap for en bedre verden

Department of Computer Science

TDT4230 - Graphics and Visualization

Scene Editor in OpenGL

Author:
Amandus Søve Thorsrud

17.04.2023

Table of Contents

The source code for this project can also be found at https://github.com/amatho/scene-editor.

1 Introduction

For the final project I have chosen to implement a basic scene editor in OpenGL, inspired by the developer tools of game engines like Unity. The camera can be moved freely around the scene by holding the **right mouse button** and using **WASD**, **space** and **left control** (pressing **shift** will move with higher speed). Objects can be spawned in front of the camera by pressing the **E key**. **Clicking** on an object will select it, opening an inspector panel where you can manipulate some of the object's properties. You can change the object's transform, 3D model, texture, and whether it is a point light or not. One of the special features of this project is the ability to change each object's vertex and fragment shaders on the fly, through an 'in-engine' text editor.

The project is implemented in Rust, using OpenGL version 4.1. Rust was chosen as it is in my experience easier to develop cross-platform programs with it. Since I wanted the project to run on macOS the OpenGL version had to be 4.1 as that is the latest supported version on that platform. In order for the project to work on my Mac I also had to develop it from the ground up, as neither the assignments from TDT4195 or TDT4230 could run on it. The scene editor consists of several parts, which are described in the following sections.

2 Entity Component System

A scene editor needs a way to represent the scene and the objects therein. In this project the scene is represented using an Entity Component System (ECS). ECS is an architectural pattern commonly used in video game development. It consists of three parts: entities, components, and systems, where an entity contains one or more components and the systems operate on the entities' components [1]. See Figure 1 for a basic overview of how ECS works. The specific ECS implementation used in this project is `bevy_ecs`, which is the ECS implementation used in the Bevy game engine [2].

Each object in the scene editor is an entity. The `Position`, `Rotation` and `Scale` components describe the entity's transform. To know how to render the entity we have a `Mesh` component, which refers to an OpenGL Vertex Array Object. When an entity is selected it has a `Selected` component, which informs the UI and renderer that an entity is selected. When the texture of an entity is changed, it is given a `CustomTexture` component that refers to the OpenGL texture ID that is to be used. A similar technique is used for editing of an entity's shaders, where a `CustomShader` component gets added. Finally we have a `PointLight` component that will make the renderer add a point light at the entity's position. These components are manipulated through the UI, which is described in the next section.

`bevy_ecs` also has support for what it calls *resources*, which are pieces of data that is not tied to a specific entity (think of global state). Some examples of *resources* in this project are `Camera`, `RenderState`, and `Input`. These can be requested by the systems that operate on components.

There are a few systems in the scene editor. The most basic ones are `move_camera`, `spawn_object`, and `select_object`. They use the *resources* mentioned above to operate on components in order to do their respective actions. The UI is implemented through two systems, `run_ui` and `paint_ui`, which will be discussed in Section 3. Finally, we have the `render` system, that actually renders the entities of the scene by reading their components, which will be explained in Section 4.

3 UI

For the UI I have used the library `egui`, which is an immediate mode GUI library for Rust. To integrate `egui` into the project I used `egui_glow`. This uses the `glow` OpenGL bindings, and is a

bit different from the bindings used in the assignments from TDT4195. `glow` is still a lower-level set of unsafe (as in unsafe Rust) bindings to OpenGL, but provides a few quality of life improvements such as string conversion between C and Rust, some type safety, and basic error reporting.

The `run_ui` system builds the UI using the *resources* mentioned in Section 2, as well as the components of the selected entity (if one is selected). This creates the layout for the inspector panel, where the user can manipulate the objects of the scene. The result of this system is used later on through the `paint_ui` system that simply calls `egui_glow`'s `paint` method. The end result when an entity is selected can be seen in Figure 2. Changing e.g. position in the inspector will directly mutate the entity's `Position` component, without needing to do anything else to update the entity's state. This is why it is so beneficial to have the UI run as a system, rather than having the UI run separate to the ECS. When the user presses the button to edit an entity's shader a full-screen text editor is shown. This uses `egui`'s built in text edit widget with a monospaced font. A visual example is shown in Figure 3.

4 Renderer

The renderer is, as mentioned earlier, implemented as a system. By using the ECS the renderer can get all the information needed in order to render a frame. The renderer uses a deferred rendering pipeline, with Blinn-Phong shading, shadow mapping, directional light, and point lights.

4.1 Shadow Mapping

The first pass of the renderer generates a shadow map. This is done by rendering from a position along the direction vector of the directional light, with the camera pointing at the center of the world, combined with a orthographic projection. A framebuffer is used as the rendering target, where a depth texture is attached. The depth values of each fragment is then stored in this texture, which will be used later in the pipeline to create shadows.

The advantage of this method over the one used in the assignments is that the shadows are more accurate, will work with any number of objects, and will correctly draw shadows from objects on top of others. However, this method also has some drawbacks that needs to be taken into account. The shadow map has a given texture size, and its orthographic projection only covers a certain area. Therefore shadow maps should usually have a 'tight' projection fitted to the scene's bounds. In my scene editor, however, there are no clear bounds, so the shadow map only covers a small area in the center of the world. Cascaded shadow mapping could be a solution to this problem. Shadow mapping also comes with some other artifacts like shadow acne and peter panning, which can be more or less solved with a correct shadow bias and tight near/far planes.

4.2 Geometry Pass

Then we actually render the geometry of the scene, but not to the screen. Since this is a deferred rendering pipeline we render the scene to a framebuffer, and write the per-fragment values needed for lighting in several textures (referred to as the G-buffer). The layout of the G-buffer used in this project can be seen in Table 1, and an example of how the G-buffer looks like in a frame can be seen in Figure 4. To find the objects to render we iterate through all the entities with `Mesh` components. Here we also check for the presence of the `CustomTexture` and `CustomShader` components, and set the texture and shader accordingly. If an object is selected, we want to render it with an outline. To do that we render the object in a slightly larger scale, and use stencil testing (more on this in Section 4.4) to discard all the fragments that overlap with the (smaller) existing object. We also set the 'selected' value in the G-buffer to inform the shaders in the deferred lighting pass that these fragments belong to an outline.

4.3 Deferred Lighting Pass

In the third pass of the rendering pipeline we will finally draw something to the screen. This is done by rendering a simple quad that fills the entire screen and sampling the textures from the G-buffer in order to calculate the lighting (i.e. color) of each fragment. When calculating the lighting we use a Blinn-Phong shading model which is very similar to the one used in the assignments, except for a new 'halfway vector' instead of the reflection vector. In addition to the point lights we already know from the assignments, we have a directional light. This is a kind of 'global' light that has the same light direction anywhere in the scene, and is the light that the shadows are meant to represent. When the 'selected' value in the G-buffer is set we skip the lighting calculations and draw an orange color, which gives a clear and bright outline. If the normal vector consists of all zeroes we know that the fragment belongs to the background (since the normals are normalized in the geometry pass they cannot be zero), and we set the color to be light blue.

The big advantage of deferred rendering is the lower performance cost of point lights. In forward rendering an object needs to be drawn once (or iterated through in a shader) for each light source, which quickly decreases performance when the number of objects and point lights increase. With deferred rendering the cost for point lights is only dependent on the number of point lights, and does not increase with the number of objects. Some disadvantages of this method are the increased memory consumption, no support for blending (this must instead be done with an additional forward rendering pass for transparent objects), and increased pipeline complexity. Deferred rendering was probably overkill for this project, but I felt like I wanted to make the rendering pipeline more interesting.

4.4 Stencil Testing and Object Selection

To be able to select an object we need to figure out which object the user clicked on based on the (X, Y) position of their mouse pointer. This is not trivial, but there are several possible solutions to this problem. One solution could be to cast a ray from the location of the mouse click in the direction of the camera, and find the first intersection with an object (using ray-OBB intersection). A second solution could be to read the contents of the depth buffer for the fragment that the user clicked on using `glReadPixels()`, and together with the X and Y coordinates unproject them back to world space coordinates. A third solution is to use `glReadPixels()` and read from the stencil buffer, which is what I have used in my project.

Stencil testing is normally used to discard fragments in a similar manner as depth testing. However, like with textures, nothing prevents us from using the stencil buffer for other purposes. In the rendering pipeline, during the geometry pass, each object is given a unique ID which is written to the stencil buffer when drawing the object. The ID is a non-zero integer, since the stencil buffer is cleared by setting all the values to zero. To know which IDs correspond to which objects, each entity gets given a `StencilId` component containing this ID. When the user clicks somewhere, we read out the value of the stencil buffer using `glReadPixels()` and compare it to each `StencilId` component. If we find a match, the entity gets given the `Selected` component. This process is done in the `select_object` system.

An advantage of my chosen method is that its pixel accurate, since it is based on values that are directly associated with the object's fragments. It is also quite easy to implement, as long as the stencil buffer is not needed for something else. The disadvantage of this method is that the stencil buffer only has a certain amount of bits per fragment, and most graphics cards only give you 8 bits. Since 0 is reserved for the background that only gives us 255 different IDs. There are some ways to get around this problem, like for example combining the values of the depth and stencil buffers to get a more unique ID.

5 Learning Resources Used

Throughout the development process I have used LearnOpenGL extensively [3]. For the shadow mapping I also found [4] and [5] extra useful. ECS was something I had learned about a couple of years ago, and the Bevy engine and `egui` are projects I knew about beforehand as a result of being interested in Rust and its ecosystem. Other than that I have also used a lot of Google, StackOverflow, and last but not least experimentation!

Bibliography

[1] Wikipedia contributors, *Entity component system — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Entity_component_system&oldid=1149188326, [Online; accessed 16-April-2023], 2023.

[2] Bevy developers, *Bevy - a data-driven game engine built in Rust*, https://bevyengine.org/, [Online; accessed 16-April-2023], 2023.

[3] J. de Vries, *Learn opengl, extensive tutorial resource for learning modern opengl*, https://learnopengl.com/, [Online; accessed 17-April-2023].

[4] Opengl-tutorials, *Tutorial 16: Shadow mapping*, https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/, [Online; accessed 17-April-2023], 2018.

[5] Microsoft, *Common techniques to improve shadow depth maps*, https://learn.microsoft.com/nb-no/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps, [Online; accessed 17-April-2023], 2020.

[6] Unity Technologies, *Ecs concepts — Entities*, https://docs.unity3d.com/Packages/com.unity.entities@0.51/manual/ecs_core.html, [Online; accessed 16-April-2023], 2022.

Appendix

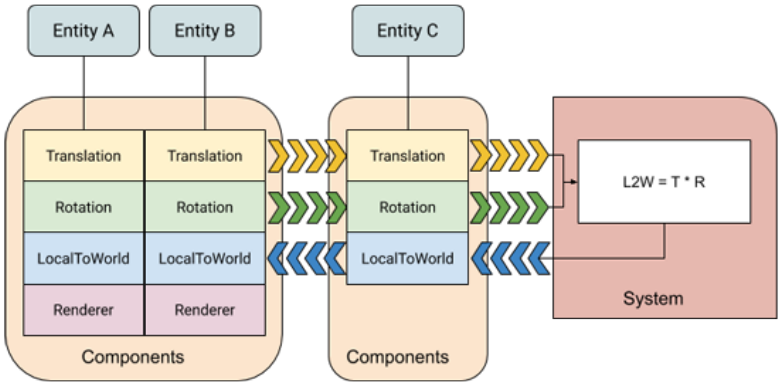

Figure 1: ECS Block Diagram. Image taken from [6].

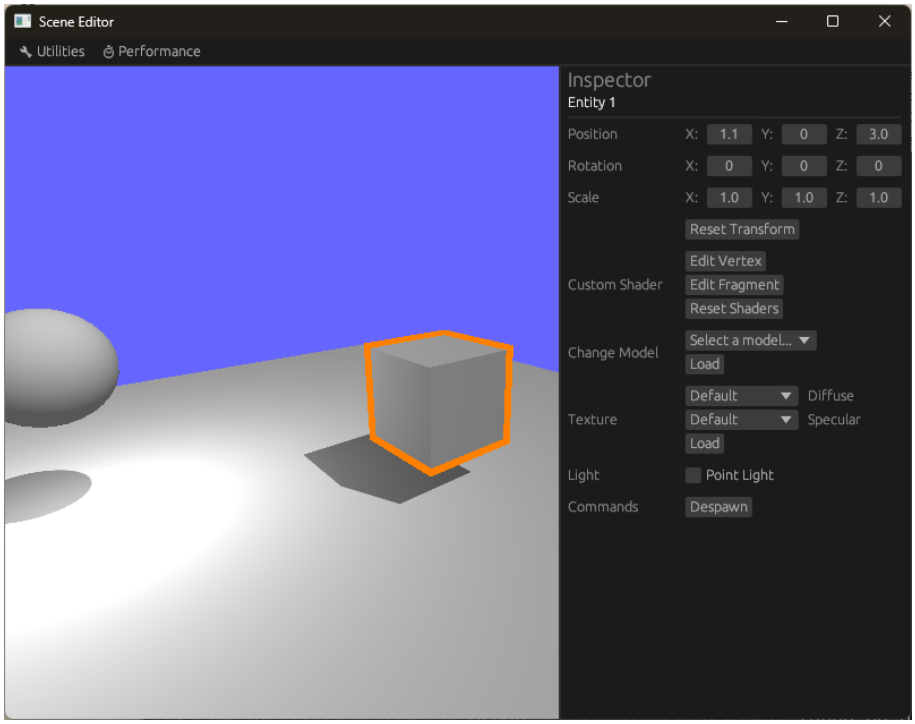

Figure 2: The entity inspector UI.

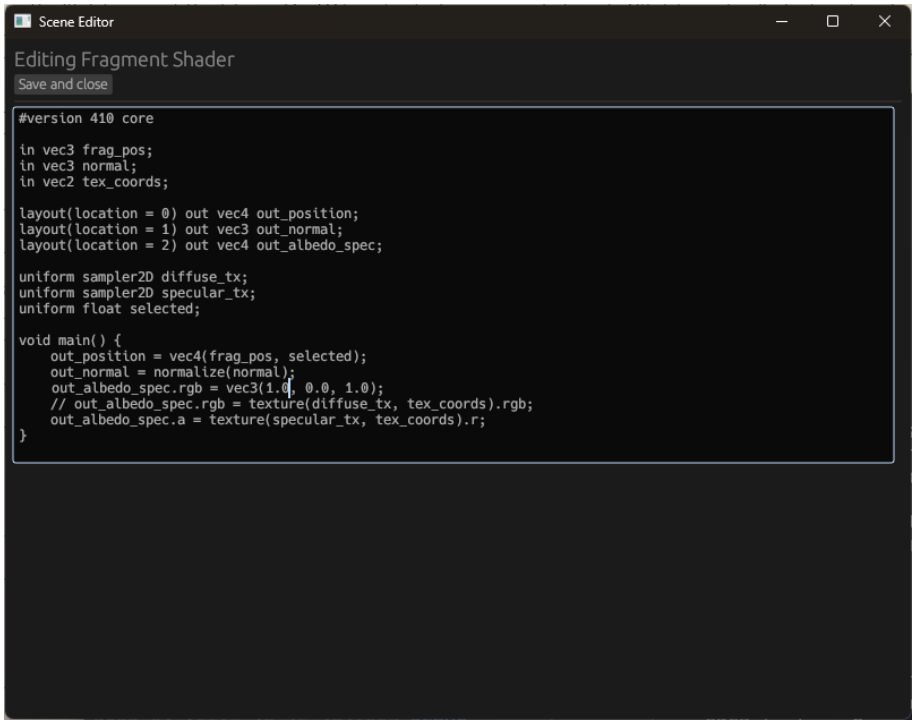

Figure 3: Editing an entity's fragment shader.

R	G	B	A
Position			Selected
Normal			
Albedo			Specular

Table 1: The layout of the G-buffer.

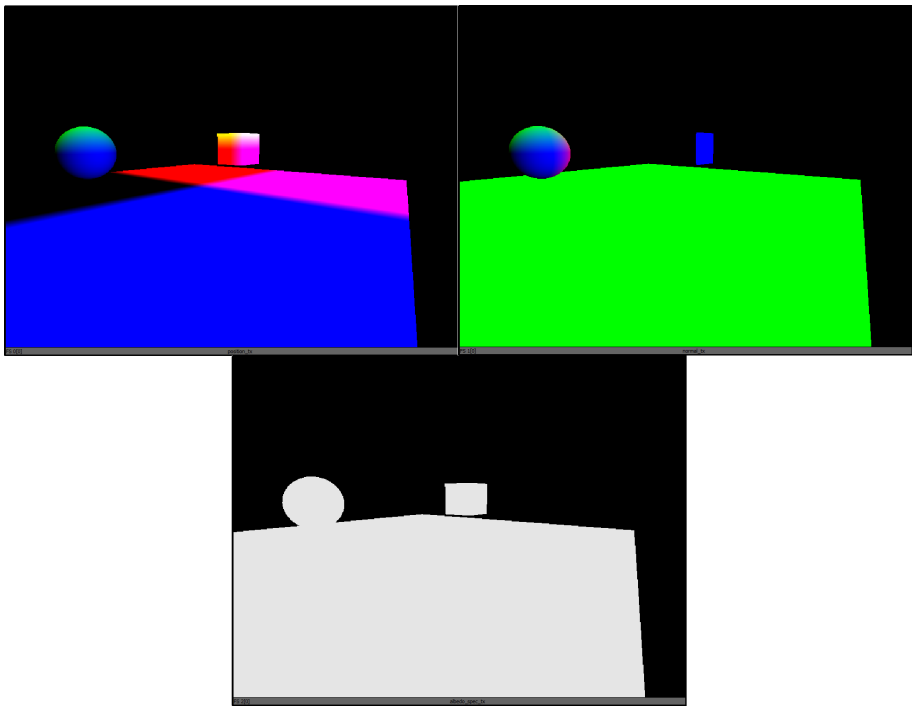

Figure 4: The contents of the G-buffer for a single frame.