OpenGL Application

David Oliver Șipoș
Group 30433

Contents

1 Subject specification

This project involves the development of a graphical application capable of rendering a 3-dimensional scene with the help of different types of light sources, animation, shadow computation and many more. The application is based on the `OpenGL API` and conceptually consists of a `C++` program running on the `CPU` complemented by vertex, geometry and fragment shaders which are executed on the `GPU`, the aforementioned library begin the glue between them.

2 Scenario

2.1 Scene and object description

The scene is a dead forest constructed with randomly placed dead trees, with a camping place and an abandoned log cabin lit by two lamp posts. There is also a hidden nanosuit in the forest in the vicinity of a blue light.

2.2 Functionalities

The scene can be inspected with a free camera with a one-time presentation animation started by the `C` key. The animation will present all aspects of the scene. The engine supports three types of lights all with the capability of shadow casting: directional light, point light and spot light. Entities in the scene are loaded from `.obj` files with material information read from an `.mtl` file. All the properties of all types of objects in the scene can be animated using keyframes, even the subcomponents of the entities. There is also support for a simple particle system with dynamic textures which change during the lifetime of a particular particle. It is possible to visualize the scene as a wireframe or individual faces with the help of the `geometry shader`. The renderer supports two modes: `normal` and `instanced`. For each entity marked normal a draw call is issued, this being only acceptable if there is a small number of entities using the same 3D model. To improve rendering speed a large number of identical entities can be marked as instanced and a single draw call will be dispatched to the video card exploiting its parallel architecture. The renderer also supports configurable skyboxes and fog.

3 Implementation details

3.1 Functions and special algorithms

3.1.1 Camera

To implement a free camera I followed this tutorial [1].

3.1.2 Loader

To improve the management of external resources I created a loader which imports: entities, shaders, skyboxes and particle textures. All objects created by the loader are dynamically allocated this class being the owner responsible of freeing the resources.

For all entities and shaders besides returning a pointer, I generate a `UUID` with the help of a header [10] which supports OS specific uuid generators. For entities using the same `.obj` file the pointer to the model class will be shared.

3.1.3 Lights and shadow mapping

The lighting calculations are very simple and use the **Phong** shading model. The color due to one light is built from three components [6]: ambient (scattered light), diffuse (light from the source) and specular (adds shine based on reflection). Directional light is easy to compute so no details will be presented. Point lights fade away as the distance increases, the decrease in amount of light is characterized by the length of the vector connecting the current fragment to the light source with the help of a second-degree polynomial attenuation. A spot light [9] is similar to a point light, it adds two additional parameters to the calculations, the light direction and the cutoff angle. To obtain the cone-shaped illumination, the angle between the to light vector and the light direction is compared to the cutoff angle to decide whether the current fragment is in the previously mentioned cone.

Shadow are calculated using shadow maps which store depth information from the light's point of view, this is involves rendering the scene from the light's perspective and storing the depth of the closest visible object. The algorithm is based on the idea that objects which are not visible to the light are in shadow. For a directional light and a spot light [2] we need to sample a simple 2D depth texture because they don't illuminate radially. In order to calculate shadows for point lights [8] we need a cube map with depth information, representing the environment around the light. The cube map is built from 6 2D textures and the obvious idea would be the render the scene 6 times. Unfortunately, this approach is rather slow. In `OpenGL` the cube map is a texture with multiple layers and we can use the geometry shader [3] [4] to generate primitives for each layer. Thus vertex data is forwarded to the geometry shader, which using 6 transformation matrices generates 6 output primitives (one for each layer specified by the `gl_Layer` built-in variable) from an input primitive and with a single draw we can render all the textures.

3.1.4 Animation system

I designed an interesting system using `C++ templates`. The animations are based on the `KeyFrame` class template which must be specialized for the classes that one wishes to animate, otherwise a compilation error will be generated because the generic class template is empty. A valid `KeyFrame` specialization must contain the attributes which are animated, a default constructor, a constructor for all attributes and finally operator overloads for multiplication with a float and addition with another `KeyFrame`, these overloads being used for the linear interpolation. The `Animation` class template stores a `std::set` of `KeyFrames` with the help of the `KeyFrameComparator` template and provides methods for starting, stopping, updating and obtaining the state of the animation and the current interpolated keyframe. The manager of all animations is the `Animator` class which is responsible for creating, deleting, starting and updating all animations. It supports two types of animations: `triggered` and `periodic`. Triggered animations are started when the `std::function` (they support lambdas which capture the context) provided at creation returns `true`, these type of animations can be marked as one time. Periodic animations have a start time and a period specified in seconds,

a period of 0 representing a one time autostarted animation. To enable animation of a class' fields besides the aforementioned `KeyFrame` template specialization the class must inherit from the `Animated` class template which provides the methods for attaching animations to an instance. This class is specialized for entities to add methods for managing subcomponent animations based on their names.

3.1.5 Particle system

A particle is basically a textured quad, represented by the `Particle` class holding its position, rotation, scale, life length, and elapsed time since creation. A particle is associated with a `ParticleTexture` which can hold at most 8 textures which will be interpolated based on the elapsed time. The `ParticleManager` holds a `std::unordered_map` binding each particle to a particle texture. The `ParticleRenderer` uses a single `VAO` to render all quads, with the help of `glDrawArraysInstanced`, by iterating over all particles in the manager and computing instance specific attributes stored in a special `VBO`. To add particle effects to scene one uses an instance of the `ParticleEmitter` class. This design is inspired from [7].

3.1.6 Rendering

As was previously mentioned the `Renderer` and `ParticleRenderer` support instanced rendering improving frame rate. The details of these approaches will be discussed in the graphics model description.

3.2 Motivation of the chosen approaches

The implemented approaches were guided by curiosity and desire for challenge. Instanced rendering was implemented first for the particle system then for the main renderer because during scene assembly the large number of triangles made the application unusable.

3.3 Graphics model

The chosen primitive used for rendering is the triangle. In what follows I will present the `VAO` format used by the renderers. To setup an instance attribute I used the `glVertexAttribDivisor` function, which has two parameters: attribute index, the number of instances for which the attribute is valid (to set an attribute for each instance one sets this paramater to 1). To force the same normal for fragments belonging to a triangle a `geometry shader` is used to assign explicitly identical normal vectors for each vertex.

3.3.1 Particle VAO

Because the particles are using instanced rendering the `VAO` contatins generic and instance-specific attributes.

Attribute	Type	Purpose
0	`vec2`	Vertex positions
1	`vec4`	Model matrix column 0

2	`vec4`	Model matrix column 1
3	`vec4`	Model matrix column 2
4	`vec4`	Model matrix column 3
5	`float`	Current texture
6	`float`	Next texture
7	`float`	Blend factor

3.3.2 Entity VAO

There two types for `VAO`s for entities based on chosen method of rendering. The one used for normal rendering is the following, attributes not present in the table are sent using uniforms:

Attribute	Type	Purpose
0	`vec3`	Vertex positions
1	`vec3`	Vertex normals
2	`vec2`	Texture coordinates

The instanced entity `VAO` looks like this:

Attribute	Type	Purpose
0	`vec3`	Vertex positions
1	`vec3`	Vertex normals
2	`vec2`	Texture coordinates
3	`vec4`	Model matrix column 0
4	`vec4`	Model matrix column 1
5	`vec4`	Model matrix column 2
6	`vec4`	Model matrix column 3
7	`vec2`	Lighting data (ambientStrength, specularStrength)

3.4 Data structures

The data structures used are the ones provided by the `Standard Template Library`. The following containers were used: `std::array`, `std::vector`, `std::set`, `std::unordered_map`. To group multiple values `std::pair` and `std::tuple` are employed throughout the source code. Custom `structs` are used for instance data which are converted to byte arrays when loading the values in the `VBO`s.

```
// for particles
struct __attribute__((packed)) ParticleInstanceData {
    float col0[4];
    float col1[4];
    float col2[4];
    float col3[4];
    int i;
    int j;
    float blendFactor;
};
```

```
// for instanced entities
struct __attribute__((packed)) MatrixColumns
{
    float col0[4];
    float col1[4];
    float col2[4];
    float col3[4];
};

struct __attribute__((packed)) EntityInstanceData
{
    MatrixColumns modelMatrix;
    float lightData[2];
};
```

3.5 Class hierarchy

Classes serving a common purpose are grouped in folders. The project has the following structure:

- animation (already presented):
  - `Animated`
  - `Animation`
  - `Animator`
- engine:
  - `Window` - represents the application window
  - `Application` - represents the application one has to extend this class to create a scene
- graphics:
  - `Renderer` - main renderer class
  - `Shader` - represents a shader; provides methods for loading activating a program and loading uniforms
  - `ShadowMap` - a wrapper around an `FBO`
- io:
  - `Keyboard` - provides information about pressed keys
  - `Mouse` - accesses mouse position
  - `Loader` - resource loader, already presented
- lights:
  - `Light` - base class for all light types
  - `DirectionalLight` - holds directional light properties

  - `PointLight` - holds point light properties
  - `SpotLight` - holds spot light properties

- objects:
  - `Camera` - holds and modifies the three vectors representing the camera
  - `Entity` - contains an object instance's place in the scene
  - `Mesh` - represents a subcomponent model
  - `Model3D` - collects multiple subcomponent
  - `SkyBox` - represents a skybox

- particles (already presented):
  - `Particle`
  - `ParticleEmitter`
  - `ParticleManager`
  - `ParticleRenderer`
  - `ParticleTexture`

- main folder:
  - `Scene` - extends the `Application` class and initializes the scene of objects
  - `main` - main file calls the run method of the scene

4 User manual

There is no graphical user interface, the camera can be controlled using the mouse and the classic `W,A,S,D` keys. At application startup the user can press `C` key to trigger an animation during which the camera flies through the scene.

5 Conclusions and further developments

In conclusion, a primitive `OpenGL` application was developed capable of rendering smaller scenes with multiple types of light, animations and visual effects. Further developments include improving the rendering pipeline, mainly making light and shadow calculations more efficient, for example I read about deferred shading. Another enhancement would increase the photorealism of the scene by implementing normal maps, reflective objects, motion blur and other post processing effects like bloom. Collision detection and the possibility of scripting are also missing from the application.

References

[1] *Camera.* https://www.opengl-tutorial.org/beginners-tutorials/tutorial-6-keyboard-and-mouse/.

[2] *Directional and Spot shadows.* https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

[3] *Geometry Shader.* https://learnopengl.com/Advanced-OpenGL/Geometry-Shader.

[4] *Geometry Shader.* https://www.khronos.org/opengl/wiki/Geometry_Shader.

[5] *Laboratory works.* https://moodle.cs.utcluj.ro/.

[6] *Light casters.* https://learnopengl.com/Lighting/Light-casters.

[7] *Particles.* https://www.youtube.com/watch?v=VS8wlS9hF8E&list=PLRIWtICgwaX0u7Rf9zkZh

[8] *Point shadows.* https://learnopengl.com/Advanced-Lighting/Shadows/Point-Shadows.

[9] *SpotLight.* https://ogldev.org/www/tutorial21/tutorial21.html.

[10] *UUID header.* https://github.com/mariusbancila/stduuid.