Design Document

OpenGL is complex due to its high level of customizability. However, the majority of users and use cases do not require extensive customizability yet are still burdened with OpenGL's complexity. Our goal is to eliminate the extensive boilerplate code and streamline and/or automate a lot of setup and teardown that have been thus far left to the user. This library makes it simple for beginners to get started with OpenGL, hence the name SimpleGL.

In this document, we will describe the design challenges we encountered, the decisions we made and why we made them as we implemented the SimpleGL library.

The Scene Class

We decided that SimpleGL would use object-oriented design to group the data of objects to be rendered and the operations on that data into classes. Furthermore, we modeled the entire scene as an object with its own operations that effect how the entire scene rendered. Scene-wide changes will be automatically applied to all objects in that scene or effect the shading and lighting in the scene. This allows the user to set up and render the full scene (including all of its objects) in one shot without the user worrying about low-level rendering details.

By implementing many of the central methods (e.g., `render()` and `add_obj()`) for the `Scene` class, we reinforce the understanding that these actions directly affect a given scene. This way of creating a Scene object and modifying it also makes it easy for us to default and hide all the OpenGL setup and teardown boilerplate in the Scene constructor and destructor. Since any use of SimpleGL will require a Scene object to exist first, all the setup will automatically be done when the user creates a Scene object prior to any work using SimpleGL, and all the teardown will automatically occur when the Scene object goes out of scope.

This encapsulation lets us set default values for the fragment and vertex shaders (a complex yet critical component of OpenGL), and the lighting and shadows, while making these components settable as well.

By making `add_obj()` a method of the `Scene` class, the scene is automatically aware of any objects that are created, and thus the object will be rendered without any extra code from the user (besides the final `render()` method call). This emphasizes the idea for the user that objects only matter when they exist with relation to the scene. A pitfall of OpenGL is that it requires the user to keep track of multiple different relations between vertex and frame buffer objects, bind and un-bind them, and so forth. In contrast to this, SimpleGL handles all these relations for the user. By specifying the type and the parameters in `add_obj()`, users can specify the details of the added object in an easy and intuitive fashion. This also allows us to do automatic cleanup when the Scene goes out of scope and to optimize the way the objects are stored in memory. See the next section for more details about objects.

Objects

Designing objects was an iterative and incremental process. We first added object functionality for built-in shapes, then added object functionality for objects loaded from .obj files, and finally, added object functionality for composite objects. These correspond to the three releases.

Built-in Shapes

We started designing objects by agreeing that it should be easy for users to add some default, common shapes that are often used in graphics programs. We used a third party library `glp` that handles one part of creating some several basic shapes. These later became our original `Shape` enums: sphere, truncatedCone, cylinder, cone, pyramid, torus, box.

Each shape creation from the third party library, however, had different parameters specific to that shape. At first we discussed and tried to implement polymorphism with the object types by creating a shape class that had subclasses like cube, pyramid, sphere, etc. The shape class would handle all the initialization boilerplate that's shared for the various objects and each subclass would have its own constructor with a unique parameter list (radius for sphere, height for cube, etc.). Each instance of shape would share a shader object, and each class would have its own render methods. Each shape subclass would also contain its own vertex buffers.

Then we realized that templatizing the shape to make the object creation more generic brought with it lots of benefits: we could easily add different types of objects later, achieve better support for composite objects, and write significantly less implementation code. So we pivoted and decided we should remove polymorphism and instead have a generic mesh class that is parameterized by an enum type `Shape` and build a wrapper around the shape creation library `glp` to only expose a generic shape creation method call.

We had originally chosen to have each instance of an object have its own vertex buffer object (VBO) and vertex array object (VAO), which are OpenGL concepts that are necessary to define an object. Later we decided to have all instances of a particular object type share their VBO and VAO because these instances could be distinguished by a simple linear transformation. For instance, a cube of side length 5 has the same VBO and VAO as a rectangular prism that has been rotated, scaled and translated. Each instance of a certain shape is represented by its model matrix, which is a composition of all the linear transformations that distinguishes each cube from the other cubes and each pyramid from the other pyramids, etc. This approach saves a lot of memory on the GPU, and is much faster than having a unique set of vertices for every single shape instance. This sharing of VBOs/VAOs will speed up the rendering time if there are lots of objects of the same type in a scene.

In order to have one VBO and one VAO per object type with different model matrices per object instance, we decided that our `Mesh` class for an object type will store one VBO and VAO, and a list of model matrices and colors for each instance of that type. These mesh objects get created via an `add_mesh()` function in the scene object. The scene object stores the mapping of object types to the respective mesh instance, which holds a VAO, VBO, and a list of model matrices and colors. The scene object then handles all the rendering instead of having the objects handle their own rendering.

.obj Objects

We then wanted to add in objects that were loaded from .obj files. This would greatly extend SimpleGL's variability because many .obj files already exist as resources for graphics programmers.

Loading objects from .obj files involves some boilerplate code and parts of that is handled by a third party library that we included. We put the calls we made to the third party code into the same uniform interface as the built-in objects above. This way, creating an .obj file object uses the same interface as creating a built-in shape object. However, then we ran up against a question: how do we determine the type of a .obj file object? We can't have them all under the same type or else we would only be able to load one .obj file and the user wouldn't be able to tell the difference between them. So we decided that .obj file objects would be uniquely identified by a string that indicates some user-defined name of that particular object. Thus, we made the `add_mesh()` function take a variant of a string and a Shape enum.

Composite Objects

And finally, we wanted to add in composite objects, where objects are made of two or more other objects.

Our design decision for this was a bit more complicated because we wanted composite objects to be treated the same way as non-composite objects. We noted that the user shouldn't have to rotate one object using a different function than another type of object and so on. They should all be able to use the same transformation functions and should all work with the same "add object" method, which should ideally remain the only way to add any object to a scene. This meant we had to change how objects were handled thus far.

To handle these issues we decided to have a base object abstract class (`BaseObj`) with a full set of transformation methods which a composite object class would then inherit and the mesh class that handles the built-in and .obj file objects would also inherit. This way, all object types would have the same interface while having different internal implementation details (since composite objects have to be transformed in a slightly different way under the hood than non-composite objects).

To access the base object class ourselves and yet hide the base object details from the user, we made an `ObjId` class which represents the id to a specific instance of a base object. The `ObjId` class has duplicates of all the transformation functions that the base object interface provides, so the user only ever has to interact with the `ObjId` s and not the bare objects themselves, and all transformations can be performed on `ObjId` instances, which the `add_obj()` method (which replaced the old `add_mesh()`) will return.

Object Creation Parameters

The design decisions for the parameters to create an object also involved some subtlety since we wanted all object creation to be performed through the single `add_obj()` call but various object creation required different parameters and we didn't want the user to have to handle long parameter lists that they might mismanage.

Therefore, we created a specific `ObjParams` struct that contains all the fields that any creation of an object might require instead of using a long parameter list.

Miscellaneous

Error Handling

We decided on using `error_conditions` and `runtime_error` which are provided by the C++ standard library in order to help the user figure out what went wrong. Particularly, we chose to bubble up errors to the user in cases where it's possible for the user to make mistakes, such as if the user accidentally submits an incorrect filepath for a .obj file, or does not specify some necessary parameter for object creation, or in cases where the underlying GLFW can error out which would impact the overall functionality of the program. User errors are likely to occur in `add_obj()` with all the parameter specification and `render()` is where many of the GLFW functions that can set an error code are called. So we set `add_obj()` to throw a `std::runtime_error` and `render()` will return a `std::error_condition` which will read the GLFW error code. Further, many of the functions that might induce incorrect user input will sanitize user input and not error out. For example, the scale functions can't take a negative scale, but instead of error out, we simply don't set the scale if the scale is not valid.

GLFW Callback Methods and Anonymous Namespace

As required by the GLFW interface, GLFW callback methods implemented in the `Scene` class have to be static. In order for the key callback to have access to object information in the scene, we had to make most variables that are associated with objects as global variables in the `simplegl.cpp` file. However, we don't want users to be able to access them since they are part of the internal implementation details. To deal with this, we used an anonymous namespace to wrap around these variables.

> Written with StackEdit and Typora.