Maintainer's Notes

See Design Document for SimpleGL design considerations.

See SimpleGL API for SimpleGL user documentation.

See SimpleGL Tutorial for a walk-through on how to write your first SimpleGL program.

Overview

In this document we will introduce the specifics about how we implemented the SimpleGL library, including the OOP design and our code style, to help future maintainers of this library keep a consistent development style.

File Organization and Style

We organize our files as follows:

- `./` : README, LICENSE, Makefile, etc.
- `include/` : contains all the required third party libraries
- `src/` : contains all the `.cpp` and `.hpp` files that make up SimpleGL
- `test/obj_files/` : contains some example .obj files
- `test/shader_files/` : contains examples of GLSL shaders
- `test/src/` : contains all the tests and demos

Please make sure that files are organized accordingly. Feel free to add subdirectories under `test/` if you want to store any other types of files (e.g. textures). If you want to split `src/` into subdirectories, make sure the Makefile is also updated accordingly.

Makefile

Our Makefile will do most things automatically, with the exception that for every new test under `test/src/` , you'll need to append it to the list of `TESTS` in the Makefile.

Below is how to use the Makefile:

- Run `make` to generate our library, libSimplegl.a, in the generated `build/` directory.
- Run `make tests` to create the test executables in the `build/` directory. You can also run `make <test_name>` to create one specific test executable.
- Run `make clean` to remove all the created files including the `libSimplegl.a` file and the created build directory and everything therein.

Feel free to update the Makefile to make it more powerful/easier to use.

Code Style

We try to stick to a consistent code style:

- use camel case for class names, enums, and typedef types
- use underscores for variable names and method names
- use all caps for constants and #include guards

Uniform style will make the code more readable, so we hope you follow these guidelines.

Scene Setup Boilerplate

If you have worked with OpenGL before, you will know that you need a multitude of method calls that start with "glfw"or "gl". And depending on the specific math library you use, you will write code using that library to deal with the linear algebra. As explained in the API, the `Scene` class handles almost all of the boilerplate related to setting up a window, shaders (along with the `Shader` class), and callbacks, among others.

When we implemented this part, we mostly referenced LearnOpenGL tutorials. These tutorials contain some initial object-oriented design like the `Shader` class, and we adapted them into our own `Shader` and `Scene` classes.

As you know from the design doc, SimpleGL aims to simplify OpenGL but at the same time this also reduces the customizability that OpenGL comes with. SimpleGL's window setup portion is pretty mature, but the shading and callback portion can still be greatly customized.

Shading

Right now shader programs are constant string literals and are directly compiled and linked at the very beginning. We default the scene to use the hardcoded shaders because programming in GLSL would add another layer of complexity and we want to free beginners from thinking about that. Shaders are also generally not where simple use cases need flexibility. We do provide the option for users to pass in their own vertex shader and fragment shader code, but there is currently no option for more advanced settings like defining a geometry shader. We could augment the current `Scene` class to allow more customizability, but we hope that these changes will keep this library simple enough for beginners.

Callbacks

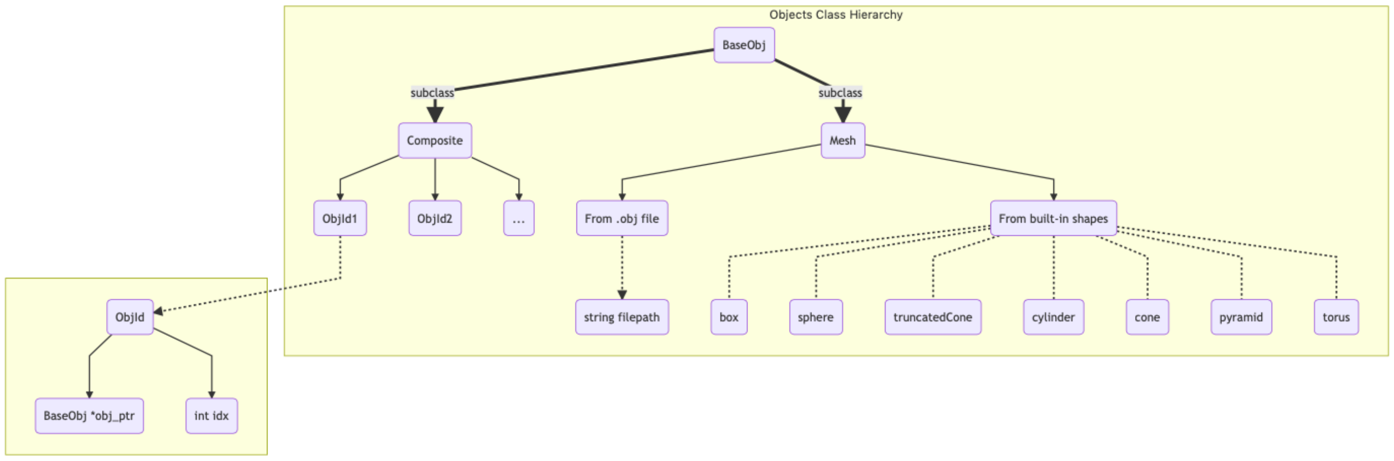
We preset for error, frame buffer size, key, mouse, and scroll callbacks. We also allow the users to pass in a callback function that will be called at the end of each render loop iteration and to choose whether to enable full key controls, but there is currently no way for them to replace these callbacks with their own. As before, we hope this could be augmented without damaging the simplicity of the library.

`BaseObj` , `ObjId` and OOP

One important aspect of SimpleGL is that it presents an abstract concept of "Object" - which is essentially a bag of triangles represented as an array of vertices - along with a set of operations that users can perform on them. We put a lot of consideration into designing this abstraction and its interface, as documented here.

`BaseObj` is the resulting abstraction, and we made it an abstract class with virtual functions. Although there is no official concept of abstract class or interface in C++, by making functions virtual, we are essentially following an object-oriented programming paradigm. Object transformations like rotations, translations, and scaling are very common in OpenGL programming, so we provide these functions so that users can directly call them without worrying about the matrix algebra. In addition, we exploit the instancing of OpenGL in the design of `BaseObj` - OpenGL can render multiple instances of the same geometry but with different transformations.

Then we created two concrete subclasses `Mesh` and `Composite` that inherit from `BaseObj` . These two classes will deal with the actual OpenGL geometry and hold the data for them (ie. the vertex arrays and transformation matrices).



Meshes

`Mesh` represents a single object entity, either loaded from .obj files or created using `glp` method calls (`glp` is a third party library that we use to help with creating geometry primitives like spheres and cubes). Each `Mesh` instance knows about its vertex buffer object and its vertex attribute object (two ideas fundamental in OpenGL), and a list of SimpleGL specific `RenderInfo` s that contains information about object colors and object transformations to be used at render time.

If you have worked with VBOs and VAOs before, you'll know they require boilerplate setup. We moved all this into the `Mesh` constructor and destructor. Right now, the types of geometry primitives you can create are limited to the ones provided by `glp` (listed here), and once a primitive is added to the scene, all subsequent instances of this shape have the same underlying vertex data. This could be undesirable when a user wants to have a pyramid of 3 sides and another of 7 sides, but most of the time it is acceptable, so we leave it as future work.

Composites

`Composite` , on the other hand, represents an object entity composed of two or more objects. It does not own any geometry data, but instead keeps track of a list of object references to `Mesh` instances - `ObjId` s. As the name suggests, when transformations are applied to a `Composite` object, the objects it owns will also transform accordingly.

Composite objects are not rendered because they are essentially groupings of `Mesh` instances; we only render `Mesh` instances in our render loop.

ObjIds

We just mentioned that composite objects would keep track of a list of `ObjId` s. In fact, `ObjId` would be the only way for users to manipulate any object in SimpleGL.

Every `ObjId` contains a pointer to the underlying object, and an id to identify which instance it is. We hide details about `BaseObj` , `Mesh` , and `Composite` and don't expose any pointers to these objects by making the two fields private. This way, the library can have full control in terms of managing the resources required for these objects.

In short, `ObjId` would be the interface that users interact with, and it should stay updated with the `BaseObj` interface.

Potential Extensions

One limitation with `BaseObj` (essentially with `Mesh`) is that we only load in vertex positions and normals data. There is no support for texture coordinates, and therefore no way to apply textures to objects, and there is no way to add other custom data associated with each vertex. To extend this possibility to the users, the following ideas could work:

1. make the `add_obj()` method virtual so that users can implement a custom way of adding objects to the scene (which involves enhancing `ObjType`, producing more than just position and normal data, etc.)
2. let users implement another subclass of `BaseObj` and pass in a custom function to `add_obj()` to create the data needed by this new subclass
3. make our glp wrapper more general so that it can handle creating objects using another library/function of the user's choice

It would not be very straightforward and some refactoring would have to happen.

SimpleGL Errors

When an error happens, we try to still carry out the task by using defaults, but if that doesn't work, we'll return an error condition with a message to help users identify the bug. As the library grows, there will be more errors that we want to remind the users about. You can add more error codes as needed.

Future Work

SimpleGL in its current state is far from being complete. Here I will discuss some future work ideas:

1. As mentioned earlier, we want to further allow different underlying data for a single type of primitives
2. Once we actually allow texture coordinate data, we could think about making a `Texture` class that does the boilerplate
3. Let the user specify scene configuration through JSON files (or any other format) and provide the functionality of storing the current scene configuration into a file
4. Provide multi-threading support

These are things that we wanted to do but didn't have time to. Feel free to add more functionality as long as it stays true the our principle of making OpenGL simple.

> Written with Typora.