Tutorial

This tutorial will walk you through the writing of a simple graphics program from start to finish. It assumes the reader has worked with C++ before. For the full source code see `test/src/hellosimplegl.cpp` in the SimpleGL repository.

Introduction

OpenGL, the most prevalent graphics specification, is used to render 3D scenes, and can be leveraged to make animations, visualizations of physics simulations, and interactive video games. However, OpenGL has a very steep learning curve due to the high level of customizability and functionality offered, even though much of this customizability isn't needed for the majority of use cases.

SimpleGL is a beginner-friendly, easy to use wrapper around GLFW, an open source implementation of OpenGL. By abstracting away and defaulting parts of OpenGL, SimpleGL significantly decreases the ramp-up time for learning how to write a graphics program compared to barebones GLFW. It dramatically decreases the lines of code needed to write a graphics program for those use cases that don't require extreme customizability.

Required Environment and Preliminary Steps

- C++17
- clang: macOS version `Apple LLVM version 10.0.1 (clang-1001.0.46.3)`
- ar (command line tool)
- glfw3: Run `brew install glfw3` if using Mac OS.

Make sure you have all of the above installed. Then, download the source code in this repository and run `make` in the top level. This will generate a `libSimplegl.a` file in the generated `build/` directory.

(You can run `make clean` if you ever need to remove all the created files.)

Create the Program

Create a file named `hellosimplegl.cpp` . Include "simplegl.hpp" in the program and set up a main function. All of SimpleGL is inside the `sgl` namespace, so we can use `using namespace sgl` to make it easier to use SimpleGL functions.

So far, the file should look like this:

```
#include "simplegl.hpp"

using namespace sgl;

int main(int argc, char** argv) {
    // to be filled in...
}
```

Set up the Scene

To start, create a `Scene` object (here we use the default values; for more options see the Scene class API):

```
Scene s; // default screen size is 800 x 600
```

Everything you want to render on the screen needs to get added to the Scene, which handles the rendering of all the objects it contains.

Add objects to the Scene

Objects in SimpleGL are identified and operated on by their `ObjId` . These objects can be one of the primitive shapes provided by SimpleGL (sphere, truncated cone, cylinder, cone, pyramid, torus, box) or a composed object (composite). Composite is a special type of object that is made of two or more objects. You can also add a custom object which can be initialized from a .obj file.

The built-in shapes, with the exception of composite, can be created with default parameters.

To add any object, you must call `add_obj()` on a `Scene` instance, which returns an `ObjId` that allows you to identify the object for transformations later. For a full explanation of all the options with respect to this function, please see the add_obj() API. Otherwise, here we will demonstrate how to use this function by example.

The only required parameter for `add_obj()` is the type of object you are creating. For example, let's add a few basic default objects:

```
// add a default sphere and default box
ObjId my_sphere = s.add_obj(Shape::sphere);
ObjId my_box = s.add_obj(Shape::box);
```

For all objects, you can also specify a non-default color using the `color` type, which is an alias for `glm::vec3` , a 3-element vector that specifies the RGB values of a color.

```
// add a default box with a color
color my_color(0.6, 0.6, 0.6);
ObjId my_box2 = s.add_obj(Shape::box, my_color);
```

Some shapes can have additional non-default parameters. For example, the pyramid defaults to have 3 sides. We can change the number of sides of a pyramid by using the `obj_params` type. This type encapsulates all the possible parameters for any kind of object, including custom objects from .obj files and composite objects, so it's a bit complicated.

`obj_params` includes the following fields:

- `accuracy, int` : used for some types of built-in objects
- `sides, int` : used for some types of built-in objects
- `filepath, std::string` : used for .obj file loaded objects
- `comp, std::vector<ObjId>` : used for composite objects

Here's an example where we create a pyramid object that has a non-default number of sides:

```
// add pyramid with 7 sides
obj_params oparams;
oparams.sides = 7;
ObjId my_pyramid = s.add_obj(Shape::pyramid, my_color, oparams);
```

Here we set the `sides` field. In order to pass in a custom `obj_params` , we do need to specify a color. However, this color value can easily be changed later for each object so for now, we can use the color object we defined already as a placeholder.

We can use the correct fields of `obj_params` to create composite objects. Let's make a composite of the default box and sphere we created above.

```
// duplicate my_sphere and my_box and combine the copies into a composite object
obj_params op_composite;
op_composite.comp = {my_sphere.duplicate(), my_box.duplicate()};
ObjId c = s.add_obj(Shape::composite, my_color, op_composite);
```

Creating a composite object is as simple as creating the objects you want to compose first and then setting the `obj_params` 's `comp` field to be a vector of `ObjId` s you want to compose and then calling `add_obj` with the parameters and the type `Shape::composite` . Here we call `duplicate()` on the `ObjId` s we want to compose because we also want to use `my_sphere` and `my_box` separately from their composition later. This way, they won't influence each other, as the copy and itself are two difference instances.

We can also load objects from .obj files using the right settings of `obj_params` and `add_obj()` . If you don't have your own .obj file you want to load, the SimpleGL repository has several .obj files to choose from in `test/obj_files/` . Pick one and get the filepath of that file. Here we chose to use kitten.obj

```
obj_params op_kitten;
op_kitten.filepath = "/path/to/test/obj_files/kitten.obj"; // replace this with your filepath
ObjId my_kitten = s.add_obj("kitten", my_color, op_kitten);
```

Transform the objects

So far we have only added objects to the scene. Now let's manipulate them by moving them to a different area of the screen, scaling, or rotating them. These transformations are accomplished via transformation functions that operate on an `ObjId` type. They are documented in detail in the ObjId Interface. Here we will explore a few transformations by example.

Let's translate and scale the pyramid we just added. Let's also set a new color for the pyramid. These translations, scale values, and color values are all arbitrary. You can change them at will.

```
my_pyramid.translate(glm::vec3(-0.2, 1.2, -0.2));
my_pyramid.scale(0.2);
my_pyramid.set_color({1.0, 0.5, 0.3});
```

For the sphere, let's similarly move it, scale it, and re-color it.

```
my_sphere.translate(glm::vec3(2, 0.8, -0.5));
my_sphere.scale(0.4);
my_sphere.set_color({0.4, 0.2, 0.3});
```

And also with the box we made earlier. Let's also rotate it 30 degrees around the axis (1,1,0).

```
my_box.translate(glm::vec3(0.1, 1, -0.8));
my_box.scale(0.2);
my_box.set_color({0.2, 0.8, 0.9});
my_box.rotate(30, glm::vec3(1, 1, 0));
```

For the second box, let's make it big enough to act as a "floor" so that we can see the shadows casted from the other objects:

```
my_box2.translate(glm::vec3(-35, -4, -35));
my_box2.scale({70, 0.01, 70}); // make this box REALLY big
```

Let's transform the composite object also.

```
c.translate(glm::vec3(1, .1, -0.2));
c.scale(0.3);
c.set_color({0.1, 0.8, 0.6});
```

And lastly, for the kitten object:

```
my_kitten.scale(0.5);
my_kitten.set_color({0.8, 0.6, 0.8});
```

Now all the objects are different colors and are at different locations, so they are all easily viewable. Their sizes are all adjusted so that different scale values can be demonstrated.

Render the scene

At this point we are satisfied with the transformations and are ready to render the scene. This is a simple one line call:

```
s.render();
```

This call will open up a window with the scene rendered when you run the program. The window will be open until you close it (you can do so by hitting the **ESC** key). You can use your mouse or the keyboard controls (explained here) to move around the camera and manipulate the view and objects.

Compile and Run the Program

Now that we have finished coding up the program, let's actually compile it and run it.

Here is the compilation command for our program:

```
$ g++ -o hellosimplegl hellosimplegl.cpp -I/path_to_simplegl/src -I/path_to_simplegl/include -
L/path_to_simplegl/build -L/usr/local/Cellar/glfw/3.2.1/lib -lglfw -lSimplegl -std=c++17
```

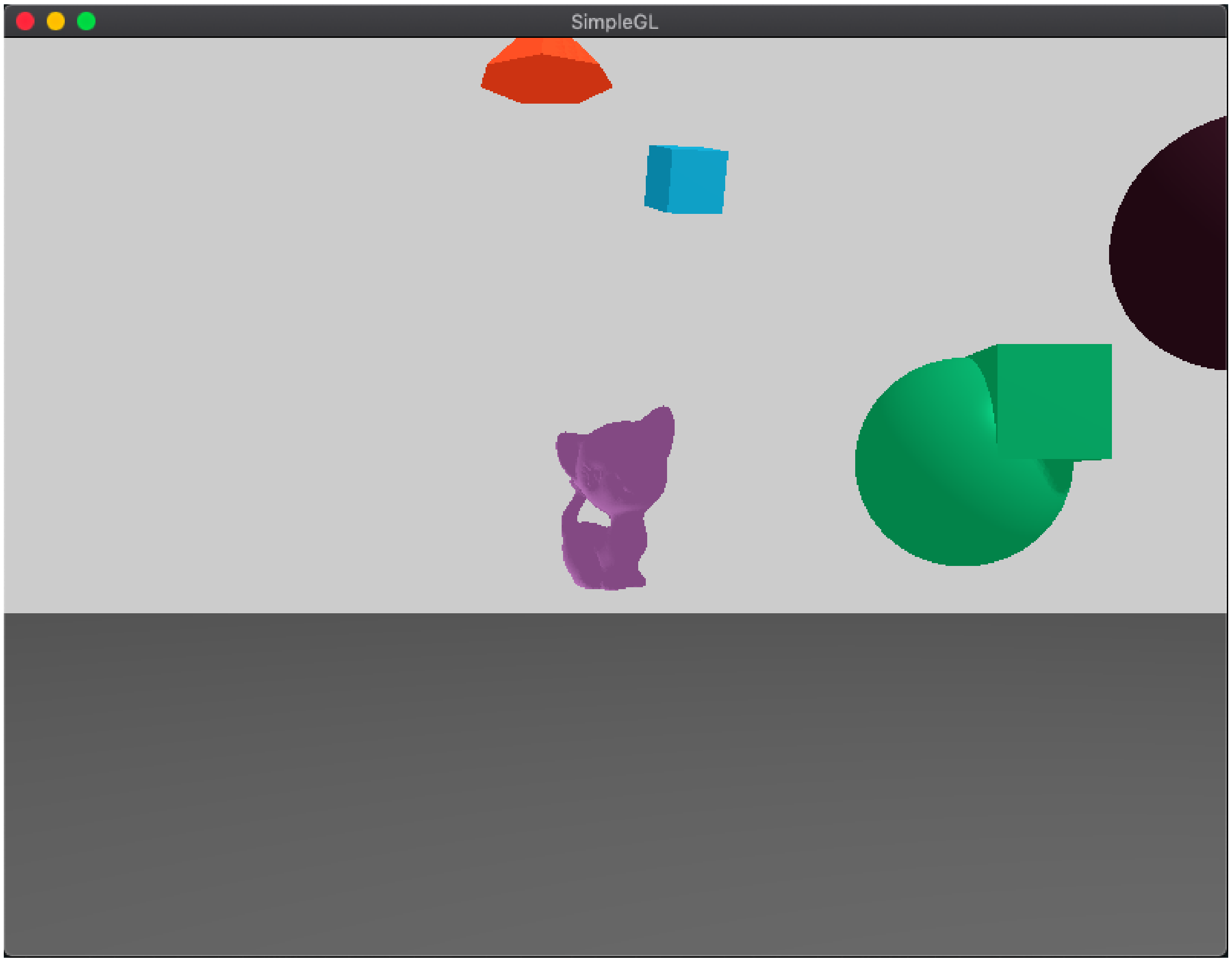
It results in an executable called `hellosimplegl`, where `/path_to_simplegl` should be replaced by the path to the SimpleGL source code repository.

The location of glfw depends on where `brew install` puts the library on your Mac.

Then to run the program, simply execute the following command in the correct directory:

```
$ ./hellosimplegl
```

A window should then pop up that looks like this:



You can move the camera around the scene either using your cursor or using the `w` (up), `a` (left), `s` (down), and `d` (right) keyboard keys. Move the view downward so that you can see the shadows the objects cast onto the "floor". Move the view upward to see the pyramid and move to the right to see the rest of the purple sphere.

For the full source code resulting from this tutorial please see `test/src/hellosimplegl.cpp` . Remember to replace the `kitten.obj` filepath with the correct filepath before compiling and executing the file.

Hooray! Now you have a running OpenGL program!

Read our API and see examples in `test/src/` to find out more about how you can use SimpleGL.

> Written with StackEdit and Typora.