SimpleGL API

Overview

SimpleGL is a simpler, beginner-friendly version of OpenGL.

This API functions as the user documentation and will detail all the methods available from the SimpleGL library and include brief descriptions and some sample code. For SimpleGL implementation details see the Maintainer's Notes.

The Scene Class

In SimpleGL, we abstract away the details of setting up OpenGL by providing the `Scene` class.

```
Scene(char *vs = nullptr, char *fs = nullptr,
      const int width = 800, const int height = 600,
      bool use_full_ctrl = false);
```

To construct a `Scene` object, which any usage of SimpleGL requires, you can either specify the various parameters yourself:

```
Scene s("my_vertex.vs", "my_fragment.fs", 300, 400, true);
```

or use the default values:

```
Scene s;
```

The argument `vs` is the filepath to the vertex shader, `fs` is the filepath to the fragment shader, `width` is the screen width of the resulting render viewbox, `height` is the screen height of the resulting render viewbox and `use_full_ctrl` determines whether or not the key control features are enabled (detailed in the last section).

The shader files are needed by OpenGL for rendering; for more information, please see the corresponding OpenGL documentation (Fragment Shader Wiki and Vertex Shader Wiki). Their construction and explanation is outside the scope of this documentation.

The `Scene` class utilizes the concept of RAII (resources acquisition is initialization):

- the constructor executes the boilerplate set-up code that OpenGL requires, including GLFW window creation and shader program compiling and linking.
- the destructor will perform all the necessary cleanup and release the acquired resources when the created `Scene` instance goes out of scope.

Objects in the Scene

In SimpleGL, the objects that can be rendered include geometry primitives, custom meshes, and composites, i.e., objects that consist of multiple other objects.

We define the enum `Shape` to be any of the following:

- sphere
- truncatedCone
- cylinder
- cone
- pyramid
- torus
- box
- composite

Among them, the first 7 values are built-in primitives, and the last value is for composite objects, which are detailed here.

Add Objects to the Scene

The `Scene` class provides a method `add_obj()` for adding an object to be rendered in the scene:

```
ObjId add_obj(obj_type t,
              const color c = {0.4, 0.4, 0.4},
              obj_params params = obj_params());
```

The first parameter is of type `obj_type`, which is an alias for `std::variant<Shape, std::string>`. For a built-in primitive or composite object, specify the `Shape` option. The `string` option acts as the unique identifier for each custom mesh loaded from a .obj file.

The second parameter is the object's color. The type `color`, which subclasses `glm::vec3`, represents the RGB values of a color. In this method, we give it a default value (a gray color) as shown above.

To create a `color` object, simply do one of the following:

```
color my_color(0.8, 0.6, 0.8); // rgb [0,1]
color my_color_255(100, 250, 230); // rgb [0,255]
color my_color_hex(0x47BECB); // rgb hex
```

The third parameter is of type `obj_params`, which holds the parameters for creating any object. It has several 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

`add_obj()` returns an `ObjId` which can be used to refer to the object just added to the scene. This is useful when we want to manipulate an object later on, but if users don't want to perform further transformations on this object, there is no need to store this return value because the object will be added to the scene regardless. If it happens that the object creation failed, an error will be thrown and needs to be handled separately.

Add a Built-in Shape

If adding a built-in shape, the two integer fields, `accuracy` and `sides`, can be set in `obj_params`:

- `accuracy` determines how accurate we want to represent an object as a geometry, and thus what the resolution of the object will be when it's rendered. This field will affect the following shapes: sphere, truncatedCone, cylinder, and cone. For other shapes, this field will be ignored.
- `sides` determines the number of sides for those built-in shapes that can accommodate a variable number of sides. Currently, this field only affects the pyramid shape. For other shapes, this field will be ignored.

Below is the sample code that constructs one sphere with default `obj_params`, and another sphere and a pyramid with different custom `obj_params`:

```
// add a sphere with default parameters
s.add_obj(Shape::sphere, my_color); // my_color defined above

// construct custom obj_params
obj_params oparams;
oparams.accuracy = 7;
// add another sphere
s.add_obj(Shape::sphere, my_color, oparams);
// add a pyramid
oparams.sides = 7;
s.add_obj(Shape::pyramid, my_color, oparams);
```

Add a Composite

To add a composite object, which is an object comprised of two or more other objects, you must first make the sub-objects separately using `add_obj()`. After we have a list of `ObjId`s returned by the previous `add_obj()` calls, we can pass that as the `comp` field in `obj_params` to combine these objects into a single composite object.

Here's an example:

```
// add the cone and the kitten mesh first
ObjId cone = s.add_obj(Shape::cone, my_color); // my_color defined above
oparams.filepath = "/path/to/kitten.obj";
ObjId kitten = s.add_obj("kitten", my_color, oparams);
// then combine them into a composite
oparams.comp = {cone, kitten}; // here we reuse oparams because comp doesn't interfere with the other parameters
ObjId c1 = s.add_obj(Shape::composite, my_color, oparams);
```

Thus, `c1` is the composite object that comprises of a kitten mesh loaded from a .obj file and a built-in cone shape. This composite object can be treated and transformed like any other non-composite object.

Note that you cannot reuse the objects you used to make the composite object unless you are willing to have both objects move together (which is usually not the case). If you already created a box and sphere, and wanted to compose them, duplicate the box and sphere before composing them.

Manipulate Objects in the Scene

By default, all the objects added will be rendered as sitting at the origin. In order to move these objects around in the scene, change their colors, scale them up or down, we can apply transformations to them. Users can apply transformations in any order; the composed transformation will have the correct ordering and produce intended results.

`ObjId` Interface

Unless otherwise specified, all object transformation functions act on an `ObjId` .

```
ObjId duplicate()
```

Duplicates an object and returns the `ObjId` of the newly created duplicate. This also duplicates the color and transformations applied to the object.

```
void hide()
```

Hides an object in the rendered scene.

```
void show()
```

Shows an object if it was previously hidden.

```
void set_color(const color c)
```

Sets the color of an object to the passed in color parameter, where the `color` type subclasses `glm::vec3` , and represents the RGB values of a color.

```
void set_model(const glm::mat4 m)
```

This function allows the user to set the model matrix manually without using the provided helper methods below. SimpleGL assumes that the user will not use any helper methods if they are setting the model matrix themselves. SimpleGL abstracts away the details of computing the model matrix, but if the user would like to handle this themselves, this function allows them to do so.

```
void reset_model()
```

Clears any transformations applied to the object.

```
void translate(const glm::vec3 translation)
```

Translates the object by the passed in translation vector where the parameter determines the (x,y,z) translation amount relative to the current transformation.

```
void set_translation(const glm::vec3 translation)
```

Translates the object by the passed in translation vector where the parameter determines the (x,y,z) translation amount where the translation is absolute with respect to the initial location.

```
void scale(const glm::vec3 factor)
```

Scales the object using the `glm::vec3` parameter that allows scaling factors to be specified for each of the three Cartesian directions (x,y,z) where the scaling is relative to the current transformation.

```
void scale(const double factor)
```

Scales the object using a singular double parameter that scales by that factor equally in all three Cartesian directions (x,y,z) where the scaling is relative to the current transformation.

```
void set_scale(const glm::vec3 factor)
```

Scales the object using th `glm::vec3` parameter that allows scaling factors to be specified for each of the three Cartesian directions (x,y,z) where the scaling is absolute with respect to the initial object.

```
void set_scale(const double factor)
```

Scales the object using a singular double parameter that scales by that factor equally in all three Cartesian directions (x,y,z) where the scaling is absolute with respect to the initial object.

```
void rotate(const float angle, const glm::vec3 axis)
```

Rotates the object the given parameter angle around the `glm::vec3` parameter axis where the rotation is relative to the current transformation.

```
void set_rotation(const float angle, const glm::vec3 axis)
```

Rotates the object the given parameter angle around the `glm::vec3` parameter axis where the rotation is absolute with respect to the initial orientation.

```
glm::vec3 get_loc() const
```

Gets the location of the object as a coordinate point `glm::vec3` form.

```
bool operator==(const ObjId& b) const
```

Overloaded `==` to allow comparison between two `ObjId` s. This allows unordered STL containers of `ObjId` to be created (one of the two components required to make a custom class hashable; the other is a custom hash function).

Other Utility Methods

```
std::ostream& operator<<(std::ostream& os, transformation& t)
```

Overloads `<<` so that transformations can be pretty printed, with each of the following printed: overall model matrix, rotation matrix, translation matrix, scaling matrix, and whether or not the object is hidden.

```
std::ostream& operator<<(std::ostream& os, color& c)
```

Overloads `<<` so that object colors can be pretty printed.

```
std::ostream& operator<<(std::ostream& os, obj_type& t)
```

Overloads `<<` so that `obj_type` can be pretty printed.

Example Usage

Below is some sample code that uses several of the above methods.

```
// translating and scaling a created object
ObjId floor = s.add_obj(Shape::box, {0.6, 0.6, 0.6});
floor.translate(glm::vec3( -35, -4, -35));
floor.scale({70, 0.01, 70});

// add a composite object of a kitten mesh and a default cone.
ObjId cone = s.add_obj(Shape::cone, my_color);
ObjId obj = s.add_obj("kitten", my_color, oparams);
oparams.comp = {cone, obj};
ObjId c1 = s.add_obj(Shape::composite, my_color, oparams);

// order of transformations doesn't matter
c1.scale(0.5);
c1.translate(glm::vec3(-1, 0, 0));
cone.translate(glm::vec3(0, 0.2, 0));
cone.scale({0.1, 0.3, 0.1});
obj.rotate(20, glm::vec3(0, 1, 0));

// add a copy by duplicating each sub-object then creating a composite
oparams.comp = {cone.duplicate(), obj.duplicate()};
ObjId c2 = s.add_obj(Shape::composite, my_color, oparams);
c2.translate(glm::vec3(1, 0, 0)); // translate it

// add a copy by directly duplicating the composite
ObjId c3 = c2.duplicate();
c3.translate(glm::vec3(1, 1, 0)); //translate it
```

Render the Scene

After loading all the objects you want into the scene, it's time to finally render the scene. Scene rendering is done by one function:

```
std::error_condition render()
```

The usage is straightforward:

```
Scene s;
// after adding objects to the scene...
s.render();
```

`render()` runs in a loop where each loop run renders the frame once. The loop is only interrupted when `glfwWindowShouldClose()` becomes true, which happens when the user tries to close the resulting render window (ie. by hitting the esc key). This GLFW function is further documented here.

In the case of an error, `render()` returns an `std::error_condition` which can be checked by the user and which will include a descriptive message about what went wrong.

Scene Utility Methods

The following functions all act on a `Scene` instance:

```
void set_callback(std::function<void(Scene *)> callback)
```

Set a user-defined function that takes in a `Scene` pointer and returns void as a callback function; this callback gets called once per render loop (i.e., once per frame). Below is an example:

The callback:

```
void print_frame_rate(Scene *scene_ptr) {
    std::cout << "Framerate: " << scene_ptr->get_frame_rate() << "\n";
}
```

And set it:

```
Scene s;
s.set_callback(print_frame_rate);
```

```
void remove_obj_all(obj_type t)
```

Removes all instances of a certain type of object from the scene, just pass in the type of object to be removed.

```
void set_light_pos(const glm::vec3 pos)
```

Sets the light position to the given `glm::vec3` which represents an (x,y,z) coordinate.

```
void set_smooth(const double smooth)
```

Sets the smoothing factor for the framerate smoothed average calculation. The smoothing factor determines how much the current frame rate affects the averaged framerate. The higher the value, the more smooth the framerate is and the smaller the effect from large changes in frame rate. To turn smoothing off, pass in 0.

```
double get_frame_rate() const
```

Returns the averaged frame rate.

```
std::chrono::milliseconds get_delta_frame_milli() const
```

Returns the last measurement of the time between two rendered frames in `std::chrono::milliseconds` .

```
void set_shadow(bool enable)
```

Toggles on/off the shadows functionality for a scene. Shadows are by default on; to turn them off, use this function and pass in `false` .

Interactive Key Controls

SimpleGL provides GLFW key callbacks that allow users to interact with the running program.

Default Controls

The following controls are always available:

W/A/S/D: move the camera up, left, down, right

Cursor: change the camera viewing direction

0: toggle the shadow rendering on/off

ESC: close the current window

Optional Controls

In the constructor of the `Scene` class, users can specify whether to enable the optional key controls, which contains the following functionalities:

Space bar: cycle through different shape categories, e.g. box, pyramid, obj1, obj2, composite

Down arrow key: move to instance level so that we can access specific instances of the current shape category

Left arrow key: select the previous instance in the current shape category

Right arrow key: select the next instance in the current shape category

Up arrow key: move to shape category level so that we can cycle through shape categories using space

H: hide the current instance (if at instance level) or all instances in this category (if at shape category level)

U: unhide the current instance (if at instance level) or all instances in this category (if at shape category level)

(the rest are all instance-level controls)

1: decrease the current instance's X position

2: increase the current instance's X position

3: decrease the current instance's Y position

4: increase the current instance's Y position

5: decrease the current instance's Z position

6: increase the current instance's Z position

7: rotate the current instance around X axis

8: rotate the current instance around Y axis

9: rotate the current instance around Z axis

P: print the `render_info` associated with the current instance; this is for showing the user what kind of manipulation would be necessary to position the objects in the scene to be the desired configuration, and potentially helpful for debugging

Error Handling

Functions where errors are possible and should be explained to the user will either return an error or throw an error. Specifically, `render()` returns a `std::error_condition` whose error message can be checked, and `add_obj()` will throw a `std::runtime_error`.

So for user level error handling, check the return value of `render()` and use a try-catch block around the code that adds objects to the scene as shown here:

```
try {
  // your code here
} catch (std::runtime_error& err) {
  std::cout << err.what() << "\n";
}
```

Errors may either be SimpleGL specific errors or expose underlying errors from GLFW (which SimpleGL uses to perform a lot of the OpenGL functionality).

Note

See Maintainer's Notes for documentation on SimpleGL implementation details.

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

> Written with StackEdit and Typora.