Name:	**Daniel Desmond Dennis**
Module:	**Computer Graphics**
Lab:	**5**
Title:	**Game Engine**
Date:	**27th November 2018**

GRAPHICS ENGINE

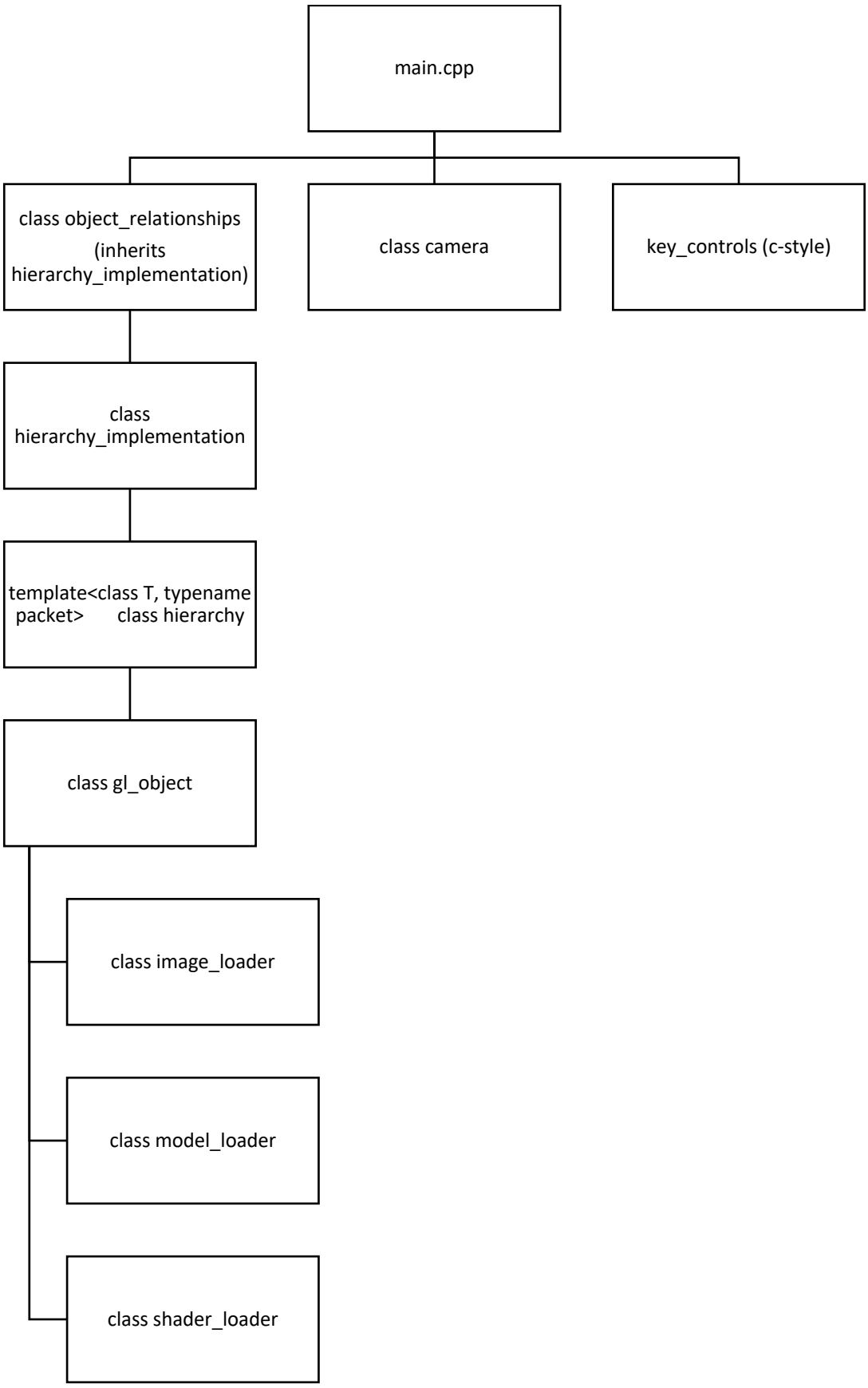
EXTERNAL DEPENDENCIES

This is written in C++11, it requires hardware support for OpenGL 3.3. The following libraries are required for compilation and linking.

- GLEW
- GLFW
- GLM
- DevIL
- Assimp

All development was done on macOS with Clang++, but no specific macOS calls are used, and so should be cross-platform. There is a Makefile included. For macOS, all of the aforementioned libraries can be installed with Homebrew. Where work for a particular class is not my own in full, or in part, this is mentioned in the corresponding header file. The general idea of this program is my own.

FILE LAYOUT Each file except for main.cpp also has a .cpp and .hpp file



GENERAL OVERVIEW

This uses GLEW to use OpenGl, and GLFW for windowing. It uses GLM for any matrix and vector operations. Models are loaded with Assimp, and textures are loaded with DevIL.

Then aim of this project was to develop a game engine, so that the code that I wrote is entirely re-usable. The scene that was demonstrated was built from a description in text files, and as such can be modified without recompiling the program. Indeed, the entire scene can be changed, with different objects and hierarchies, and different key presses to interact with the program.

The class `hierarchy <gl_object, packet_t>` is a class template that holds a data structure of an object hierarchy, each node is of type `gl_object` (which handles all of the graphics calls related to a particular object), this class calls `image_loader` which loads textures to the GPU, `model_loader`, which loads the vertices and normal to the GPU, and `shader_loader`, which loads and compiles the fragment and vertex shader for each object. The `gl_object` class then draws the object in the game loop every time the `render()` function is called. `packet_t` is a datatype in the program for sending commands to certain nodes in a hierarchy, this is explained in detail further down.

The `hierarchy_implementation` class builds the hierarchy based on a text file describing the hierarchy, and creates multiple instances of `hierarchy<gl_object>` as appropriate, it pushes new children into it the hierarchy as appropriate.

The class `object_relationships` reads in actions from another text file. It inherits the functionality of `hierarchy_implementation`. Each line of this text file has the object id, the key press associated with it, the action to take (for example move the object on the x-axis, rotate on the y-axis), and a scalar value to determine how much to move by. It then saves these actions in a `std::vector` of a `struct` which holds the object id, a function pointer for the type of action that it will take, and the scalar value specified. The following is what the key press action for ‘j’ looks like.

```
typedef struct action
{
	void (*which_function)(node_id_t, float, hierarchy<gl_object, packet_t>*);
	float step;
	node_id_t address;
} action_t;

std::vector<action_t> key_j_actions;
```

From the GLFW keypress handler, it calls the following function with the appropriate enumerator for the key press type. The following is a shortened version which just implements the ‘j’ key press:

```
void object_relationships::key_handler(key_press_t keypress)
{
	switch(keypress)
	 {
	case key_press_t::key_j:
	     for(int i = 0; i < key_j_actions.size(); i++)
```

```
            {
                (key_j_actions[i].which_function)(key_j_actions[i].address,
key_j_actions[i].step, objects[(key_j_actions[i].address)[0]]);
            }
        break;
       }
}
```

This executes a function, pointed to by `which_function`, passing in the step, the id of the child to push the action to, and a pointer to the hierarchy to send thid action to.

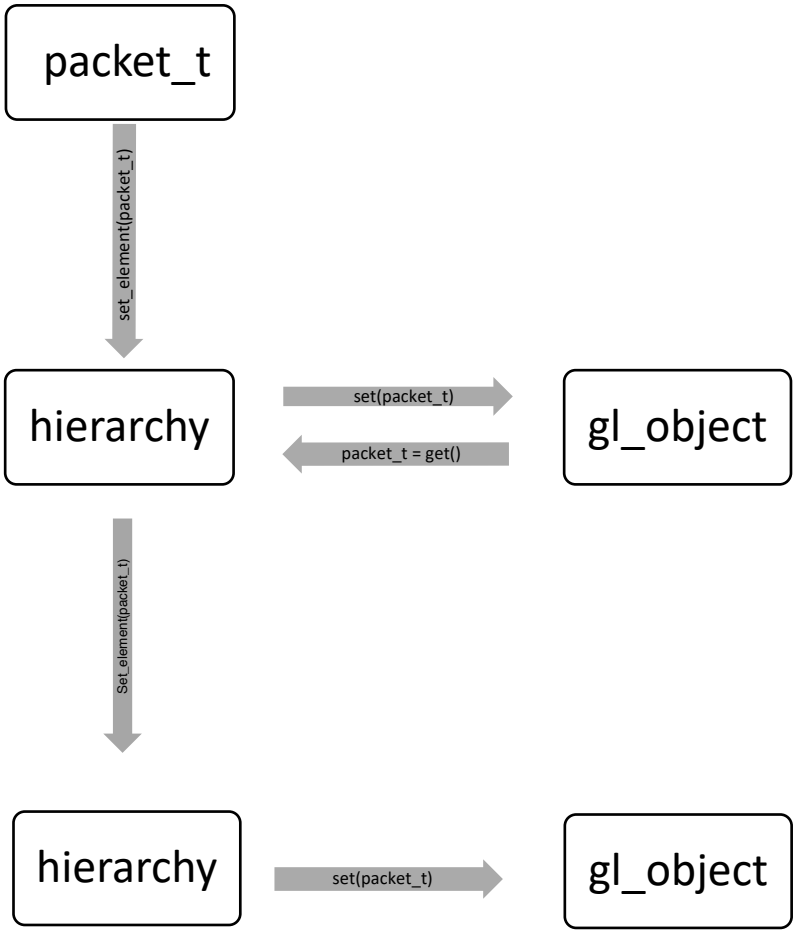
The hierarchy class is a class template, which holds the gl_object class. It holds four methods for interacting with the objects. The first, `create_family` is used to initialise the hierarchy, and sets up the first node. Subsequent calls can be made to `insert_element` to add children, an id is passed in to specify where the child will go, so if one wanted to make a new child at with id 3,4,2,5, the id 3,4,2, is passed in to `insert_element` to make a new child there (in this example, because the id of the new child being inserted is 3,4,2,5, there are already five children there, with ids: 3,4,2,0, 3,4,2,1, 3,4,2,2, 3,4,2,3, 3,4,2,4 respectively). Then actions can be pushed to a node in this hierarchy using `set_element`, or to the entire family of that hierarchy using `broadcast`. The `broadcast` method just passes the same data to every node in the hierarchy. `set_element` moves a particular child in the hierarchy, and then affects the change appropriately on the children. Messages are passed in a struct of type `packet_t` as per below. There is an enumerator to determine what the action is, and the actual data is pointed to in the data variable, so anything, including, something like a view matrix can be passed in this way.

```
typedef struct packet_t
{
    // Any changes to this enum must be reflected in:
    // packet_t::mask_t object_relationships::return_mask(char* str)
    // void gl_object::set(packet_t data)
    typedef enum mask
    {
        empty,
        view,
        persp,
        model,
        coordinates,
        global_view,

        move_x,
        move_y,
        move_z,

        rotate_x,
        rotate_y,
        rotate_z,

        render,
        compile,

        light_pos,
        viewer_pos,

        scale
    } mask_t;

    mask_t mask;
    void* data;

    packet_t(mask_t mask_in){ mask = mask_in; }
} packet_t;
```

In `gl_object`, when an action is passed in a `packet_t` variable, it has a switch statement to handle the enumerated key types, for example

```
case packet_t::move_y: view = glm::translate(view, glm::vec3(0.0f, *(float*)data.data, 0.0f));
return;
```

An example of how a move_x action would be processed in the hierarchy is as follows. The hierarchy class would locate the object that is to be moved, perform the move in `gl_object` of that node by passing the `packet_t` variable into that `gl_object`, then call `get()` to `gl_object` which changes the `packet_t` to have the result of a view * model operation which is then propagated to all children.



The general layout of the hierarchy setup file is as follows

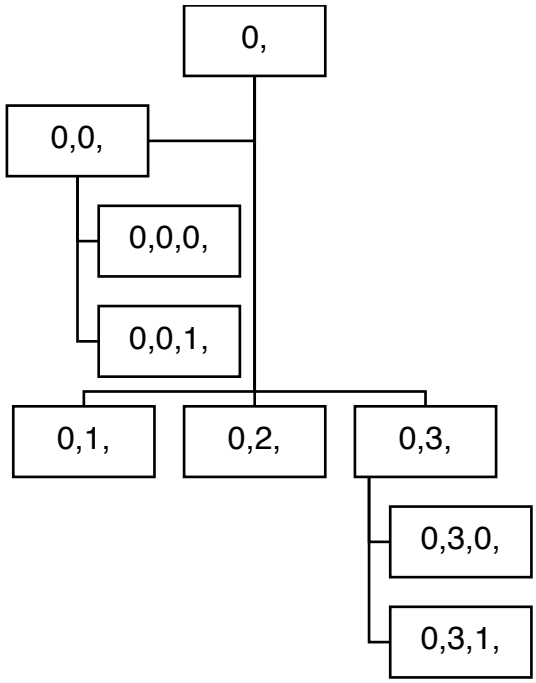
```
[id] [path to model] [path to texture] [path to vertex shader] [path to fragment shader] [initial coordinates]
```

An example of how a hierarchy can be setup is as follows

```
0, model1.dae path/to/vertex_shader1.vs path/to/fragment_shader.fs 0.0f 1.0f 30.0f
0,0, model2.dae path/to/vertex_shader1.vs path/to/fragment_shader.fs 0.0f 1.0f 0.0f
0,0,0, model3.dae path/to/vertex_shader1.vs path/to/fragment_shader3.fs 0.0f 2.0f 0.0f
0,0,1, model1.obj path/to/vertex_shader1.vs path/to/fragment_shader.fs 20.0f 1.0f 0.0f
```

```
0,1, model1.blend path/to/vertex_shader2.vs path/to/fragment_shader2.fs 5.0f 1.0f 0.0f
0,2, model1.dae path/to/vertex_shader1.vs path/to/fragment_shader.fs 0.0f 1.0f 10.0f
0,3, model1.dae path/to/vertex_shader3.vs path/to/fragment_shader.fs 0.0f 8.0f 0.0f
0,3,0, model1.dae path/to/vertex_shader1.vs path/to/fragment_shader4.fs 0.0f 1.0f 0.0f
0,0,1, model1.dae path/to/vertex_shader1.vs path/to/fragment_shader.fs 10.0f 1.0f 0.0f
1, model1.dae path/to/vertex_shader1.vs path/to/fragment_shader.fs 0.0f 1.0f 0.0f
1,0, model1.dae path/to/vertex_shader1.vs path/to/fragment_shader.fs 0.0f 1.0f 0.0f
```

A diagram of hierarchy ‘0’ is described in the following diagram.



Actions can then be specified in another file as follows. The `loop` and `init` key types are special cases, `init` will run once on setup and `loop` will run every time the program goes through a loop.

In general

```
[id] [key press] [action] [scalar amount]
```

For example

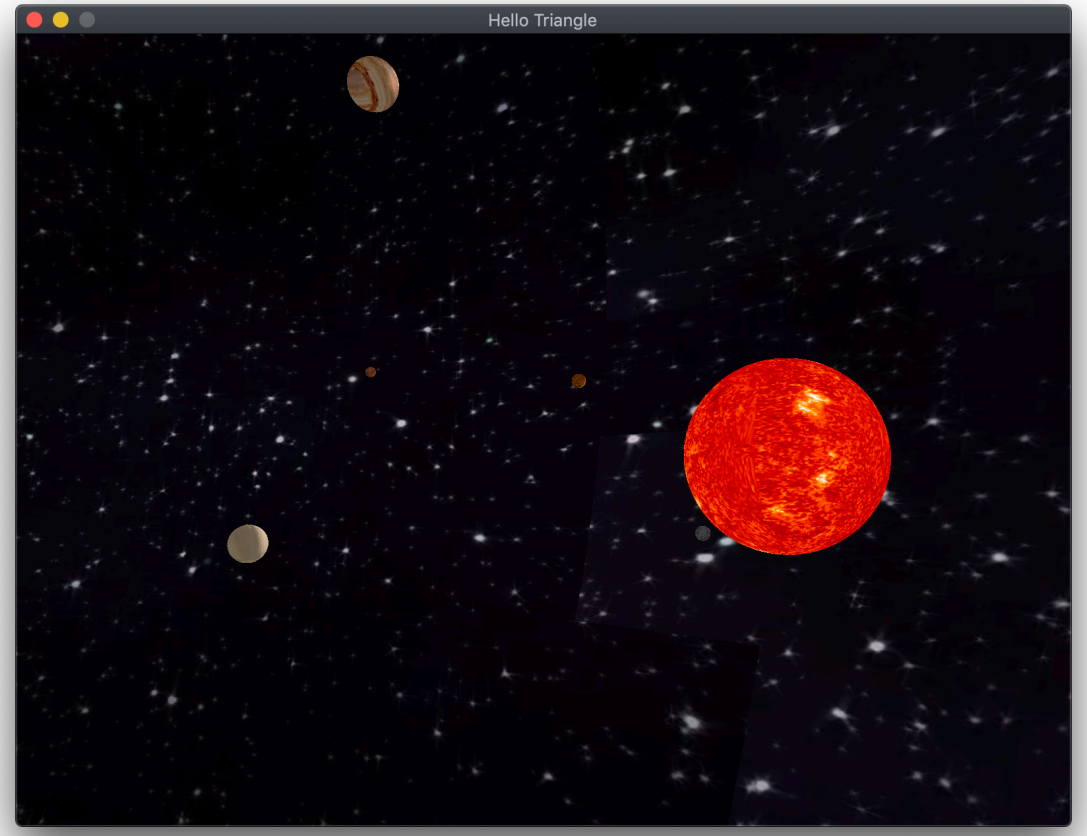
```
0, init scale 2.0f
0,0,1, init scale 0.5f
0, loop rotate_x 0.001f
0, key_up move_y 1.0f
0, key_down move_y -1.0f
0, key_right move_x 1.0f
0, key_left move_x -1.0f
0, key_fullstop move_z 1.0f
0, key_solidus move_z -1.0f
```

In this example then `0,` will be scaled once in the setup to twice its sizem and `0,0,1,` will be scaled to hald its size. `0,` (and all of its children by definition of the hierarchy) will rotate on the x axis indefinitely. If I press the down key, the object will move down, etc.

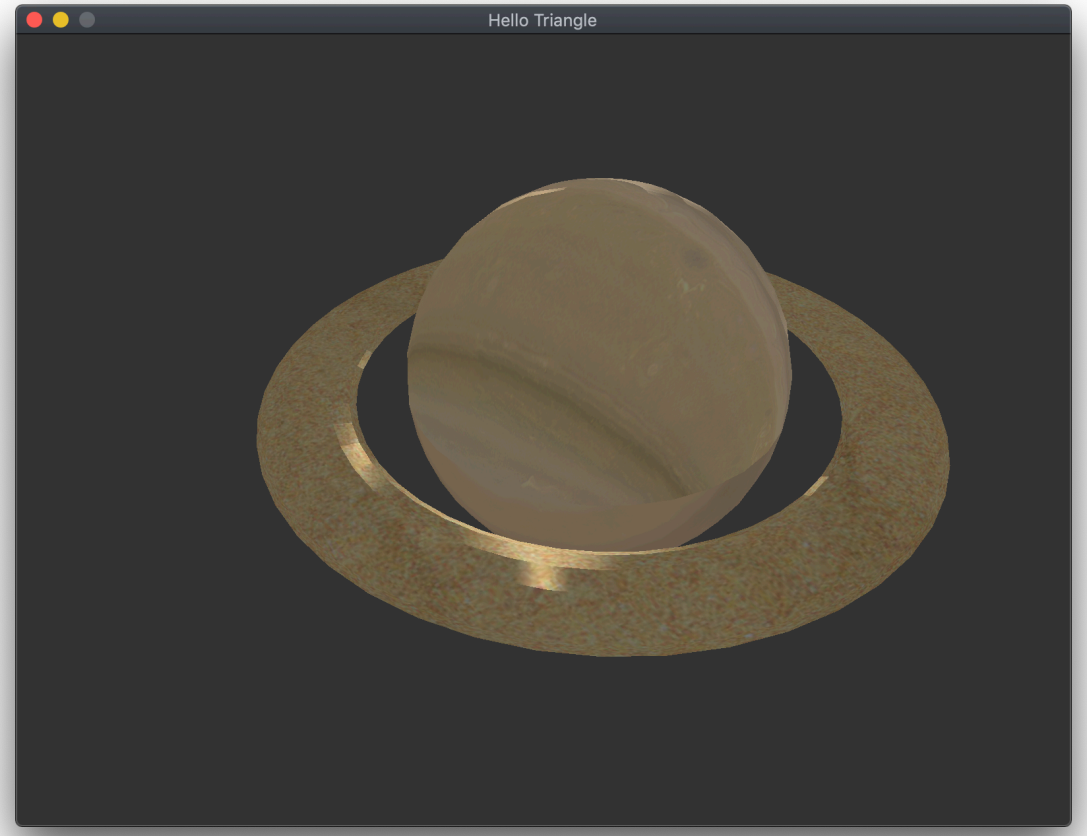
The camera class implements a basic camera, clicking the mouse on the screen will lock the mouse and it can be used to pitch and yaw. The *WASD* keys move through the 3D world.

DEMONSTRATION

To demonstrate this working, a wildly inaccurate mock-up of the solar system was modelled with the sun at the centre, and planets circling around at different rotational velocities. To have planets rotating around at different rotational velocities, the program has a feature where a virtual object in a hierarchy can be specified (by putting *null* for the model, texture, vertex shader, and fragment shader files), this means that the image_loader, shader_loader, and model_loader files are not used (they are dynamically allocated), thus saving memory and processing time, they can still be manipulated like a normal object, the results are only visible however if it has children. Earth also has a its moon rotating around it, which itself it rotating around the sun. All models used here were made in Blender.



The following is what Saturn looks like close up, (set up in a different file). The ring is a separate object that is scaled appropriately before the program loop starts. Note that the Phung illumination model has been implemented in Saturn's fragment shader.



The setup file for this looks as follows (comments are denoted with #)

```
# Universe background
0,  ancillary/models/new_ball.dae  ancillary  /textures/universe.png  ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 0.0f 0.0f
# The Sun
1    ancillary/models/new_ball.dae    ancillary/textures/sun.jpg    ancillary/shaders/core.vs
ancillary/shaders/core.frag 0.0f 0.0f 0.0f
# Mercury
2, null null null null 0.0f 0.0f 0.0f
2,0  ancillary/models/new_ball.dae  ancillary/textures/mercury.jpg  ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 10.0f 0.0f
# Venus
3, null null null null 0.0f 0.0f 0.0f
3,0   ancillary/models/new_ball.dae   ancillary/textures/venus.gif   ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 20.0f 0.0f
# Earth and moon
4, null null null null 0.0f 0.0f 0.0f
4,0   ancillary/models/new_ball.dae   ancillary/textures/earth.jpg   ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 40.0f 0.0f
4,0,0   ancillary/models/new_ball.dae   ancillary/textures/moon.jpg   ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 0.0f 0.0f
# Mars
5, null null null null 0.0f 0.0f 0.0f
5,0   ancillary/models/new_ball.dae   ancillary/textures/mars.jpg   ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 50.0f 0.0f
# Jupiter
6, null null null null 0.0f 0.0f 0.0f
6,0  ancillary/models/new_ball.dae  ancillary/textures/jupiter.jpg  ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 60.0f 0.0f
# Saturn
7, null null null null 0.0f 0.0f 0.0f
7,0    ancillary/models/ball.dae    ancillary/textures/saturn.jpg    ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 5.0f 70.0f 0.0f
```

```
7,0,0,    ancillary/models/ring.dae    ancillary/textures/sand.jpg    ancillary/shaders/cube.vs
ancillary/shaders/cube.frag 0.0f 0.0f 0.0f
```

The actions file looks like this

```
0, init scale 500.0f
1, init scale 10.0f
6,0, init scale 5.0f
2,0, init scale 0.9.0f
4,0, init scale 1.5f
4,0,0, init scale 0.5f
2, loop rotate_x 0.00005f
3, loop rotate_x 0.00003f
4, loop rotate_x 0.00007f
5, loop rotate_x 0.000034f
6, loop rotate_x 0.000021f
4,0 loop rotate_x 0.0034f
7, loop rotate_x 0.000043f
7,0, init scale 4.0f
7,0,0 init scale -40.0f
```

APPENDIX

The following are valid key presses that can be specified in the actions file.

```
key_up
key_down
key_left
key_right
key_comma
key_fullstop
key_solidus
key_a
key_b
key_c
key_d
key_e
key_f
key_g
key_h
key_i
key_j
key_k
key_l
key_m
key_n
key_o
key_p
key_q
key_r
key_s
key_t
key_u
key_v
key_w
key_x
key_y
key_z
```

The following are valid action types that can be specified in the actions file

```
rotate_x
rotate_y
rotate_z
move_x
move_y
move_z
scale
```

All texture formats supported by the DevIL library are supported with this program. All model formats supported by the Assimp library are supported with this program.