Table of Contents

Getting Started

The project is available on github

- Project (https://github.com/hen3-steenberg/COS3712)
- Release Artifacts (https://github.com/hen3-steenberg/COS3712/releases/tag/V_2.0.0)
- Documentation (https://hen3-steenberg.github.io/COS3712/).

This section explains how to set up your environment and build the project.

1. Dependencies

To make the build process more repeatable most dependencies are included in the project as submodules (The files are in this project). There are two major exceptions.

1.1 A c++ compiler

More specifically the project uses the C23 `#embed` feature to embed resources such as shaders / objects / textures into the executable. At the time of writing both the `clang` and `gcc` toolchains support this feature.

Since the `clang` compiler is available on all major desktop platforms the project is configured to use the clang compiler if it is detected (`CMakePresets.json`).

To see if other compilers support this feature, see the C23 Compiler support (https://en.cppreference.com/w/c/compiler_support/23) website.

1.2 Vulkan SDK

The vulkan sdk is just too large to include in the project and needs to be installed manually.

If you use a package manager, the sdk should be available on most and could be installed this way.

Otherwise, go to the Vulkan SDK Download (https://vulkan.lunarg.com/sdk/home) page and download the installer for your operating system.

2. Build

2.1 List the presets for your system

```
cmake --list-presets
```

The output will look something like

```
Available configure presets:

  "linux-debug"   - Linux Debug
  "linux-release" - Linux Release
```

The ID of the preset is the quoted string. Pick a preset and remember its ID that id will be referenced as `<preset-id>` going forward.

2.2 Create a subdirectory

```
mkdir build && cd build
```

The following instructions will assume that you are in a subdirectory of the project. If you want to use a different name for this directory you are welcome.

2.3 Generate the build files

```
cmake -S .. -B . -G Ninja --preset=<preset>
```

As an example if I wanted to create a debug build on linux the command would be.

```
cmake -S .. -B . -G Ninja --preset="linux-debug"
```

If this command executes successfully all dependencies have been found by cmake.

If it is needed to try again the command must be modified a bit.

```
cmake -S .. -B . -G Ninja --preset=<preset> --fresh
```

2.4 Build the project

```
cmake --build ./
```

2.5 Run the project

Linux

```
./cos3712
```

Windows

```
./cos3712.exe
```

Dependencies

As mentioned in the Getting Started (Getting Started) section most of the dependencies are already submodules of the project. This section explains exactly what those dependencies are and where they are used.

1. Obscure

This is the main dependency of the project. Obscure is a rendering engine that I wrote from scratch. It aims to provide the boilerplate code needed to use vulkan while making it easy to add custom graphics pipelines.

1.1 Vulkan SDK

Vulkan SDK (https://www.lunarg.com/vulkan-sdk/) The vulkan API is central to the project and is the API through which the various graphics pipelines are executed and ultimately displayed.

The vulkan sdk also supports the executable `glslc` which is used to compile all glsl shaders. These compiled shaders are easier to interpret by GPU drivers making the graphics pipeline consistent across a wide variety of GPU's.

1.2 GLFW

GLFW (https://github.com/glfw/glfw) This library manages the output window / screen as well as any mouse / keyboard input.

1.3 VulkanMemoryAllocator

VulkanMemoryAllocator (https://github.com/GPUOpen-LibrariesAndSDKs/VulkanMemoryAllocator) In vulkan it is possible to allocate memory on the GPU being used. This single memory allocation can then be used by various buffers or textures. Memory allocation strategy however, is a very large / specialized topic, thus I opted to just use an industry standard library to manage this aspect of vulkan for me.

1.4 glm

glm (https://github.com/g-truc/glm) This library defines C++ types for all of the intrinsic types that is available in the glsl language. They can be used to do operations on data

that will be sent to the GPU and will behave in a similar way.

1.5 stb

stb (https://github.com/nothings/stb) In particcular `stb_image.h` is used to load images from various format to a format that is usable for vulkan.

2. rapidobj

rapidobj (https://github.com/guybrush77/rapidobj) The rapidobj library is used to parse Wavefront (`.obj + .mtl`) files which allows the project to use assets generated in Blender (Blender projects can be exported to the Wavefront format).

3. glNoise

glNoise (https://github.com/FarazzShaikh/glNoise) glNoise is a glsl library that provides various functions for generating pseudo random noise in shaders.

The floor is generated using perlin noise from this library.

The library is primarily for use in a typescript project, but it is possible to reference the glsl code directly.

Overview

This section is an overview of the files in the project and their purpose. The project itself consists of a few directories, each with a specific purpose.

src

`src` **(src)** The `src` directory contains all of the source files that is compiled into the final program. This includes resources in their final states.

2. shaders

`shaders` **(shaders)** The `shaders` directory contains the source code for all of the shaders used in the program.

3. objects

`objects` **(objects)** This directory contains the Blender (`.blend`) files for various objects that needs to be drawn when the program is run.

4. obscure

`obscure` **(obscure)** This directory contains the source for the obscure library.

5. 3rd_party

The `3rd_party` directory contains the sources for most of the dependencies required by the project.

See the dependencies (Dependencies).

6. Writerside

The `Writerside` directory contains the source files for this document.

src

Following is a quick overview over the C++ files in this directory.

1. `cos3712.cpp` This is the entry point and where everything is brought together.
2. `Animation.cppm` A module with all of the programming for different animations and combining them.
3. `Building.cppm` The code for tying together building models and the locations where they should be drawn.
4. `Camera.cppm` This module interprets user input and translates it into the appropriate camera movement.
5. `Drone.cppm` The module for animating and managing the drones.
6. `Floor.cppm` This module defines the graphics pipeline for drawing the floor.
7. `GlobalState.cppm` This module provides shared access for variables that is needed in more than one other module.
8. `MenuOverlay.cppm` Provides the functions to draw the various menus and overlays and respond to any events that is triggered through the menu.
9. `ObjectPipe.cppm` The code defining the pipeline for drawing different objects. This includes Buildings, Vehicles and drones.
10. `Resources.cppm` + `resources/Resources.cxx` This is the module in which the resources are embedded at compilation time. It provides functionality to access these resources.
11. `Vehicle.cppm` Code for sharing the same model between different different vehicles with each having a distinct transformation.

There is also a `resources` subfolder containing all of the resources that is embedded in the application in their final processed state.

1. `objects` Each of the blender objects (`.blend`) are exported in the wavefront format

which exports an object (`.obj`) file and a material file (`.mtl`)

2. `shaders` Contains all of the shaders that the application use compiled to the SPIR-V (`.spv`) format.
3. `textures` Contains all of the textures used by the application, currently empty.

shaders

The code for both the vertex (`.vert`) and fragment (`.frag`) shaders, written in the `glsl` language, for the following entities can be found in this directory.

- object
- floor

The glNoise shader library is located in the `glNoise` subdirectory.

The instructions for compiling the shaders is located here in the `CMakeLists.txt` file.

The shaders are compiled to `src/resources/shaders`.

objects

This folder contains the blender (`.blend`) files for all of the objects in the application except for the floor which is generated programatiacally.

1. `Building1.blend` A thin tower like building, provided by my wife.
2. `Building2.blend` A large multistory building, provided by my wife.
3. `Building3.blend` A simple house, provided by my wife.
4. `drone.blend` A simple torus, cone and beam to model a drone.
5. `hole_building.blend` A large regtangular building with a cylindrical hole near the top.
6. `portal_building.blend` A building building with a large circular 'entrance' through which vehicles can enter or exit the scene out of view of the user.
7. `ship.blend` A spaceship like model that is used as a flying car.

obscure

Obscure is my rendering engine which I wrote.

The main aim or differentiator of this rendering engine is that it assumes that the user wants to customize the pipelines that the application will use and tries to make it very easy to define custom pipelines.

To define a custom pipeline the user has to define a structure with specific members.

1. shader_list typename

A compile time list of shaders that the pipeline requires.

A template specialization of the struct `shader_loader` is required for each shader in the list. The `shader_loader` specialization should provide a static method `load_shader()` which returns the data of the compiled shader as something which can be converted into a `std::span<const uint32_t>` the concept `obscure::vulkan::shader_data` is provided to confirm the return type is valid at compile time.

Examples

```
export struct ObjectPipe {
    using shader_list = obscure::make_set<
        resources::shader_name::object_vertex,
        resources::shader_name::object_fragment
    >;
    ...
    };
```

From `src/ObjectPipe.cppm` lines 68-72.

```
export
template<resources::shader_name shader>
struct shader_loader<shader> {
    static obscure::vulkan::shader_data auto load_shader() {
        return resources::get_data<shader>();
```

```
    }
};
```

From `src/Resources.cppm` lines 76-82;

2. initialize function

The `initialize` function handles the runtime initialization of the pipeline. The `initialize` function should return a value of a type that satisfies the `obscure::vulkan::pipeline_builder` concept.

Obscure provides a function `obscure::vulkan::default_pipeline_builder`, that initializes all of the fixed function configuration options for the pipeline.

```
auto result = obscure::vulkan::default_pipeline_builder<1, 2,
    vk::PrimitiveTopology::eTriangleList,
    vk::PolygonMode::eFill,
    vk::FrontFace::eCounterClockwise,
    vk::ShaderStageFlagBits::eVertex,
    vk::ShaderStageFlagBits::eFragment>
(
    render_pass,
    samples,
    shaders,
    {
        vk::VertexInputBindingDescription{
            0,
            sizeof(ObjectModel::vertex_t),
            vk::VertexInputRate::eVertex
        }
    },
    {
        vk::VertexInputAttributeDescription{
            0,
            0,
            vk::Format::eR32G32B32Sfloat,
            offsetof(ObjectModel::vertex_t, position)
```

```
        },
        vk::VertexInputAttributeDescription{
            1,
            0,
            vk::Format::eR32G32B32Sfloat,
            offsetof(ObjectModel::vertex_t, normal)
        }
    });
```

From `src/ObjectPipe.cppm` lines 80-110.

Further configuration is then left to the implementer.

2. draw_calls typename

The `draw_calls` type must inherit the `obscure::vulkan::draw_call_base`. This struct contains all of the code for invoking the graphics pipeline with various parameters. Any public functions of the `draw_calls` type is exposed in the `frame_ctx`. The `frame_ctx` is obtained by calling `begin_frame()` on the `graphics_context`.

Assignment 2

This section is about all of the requirements for assignment 2, and how they were implemented.

1. City Layout and Buildings

This section was broken up into 2 pieces.

1. Buildings (Buildings)
2. Floor (Floor)

2. Cars & Drones

This section was broken up into 3 pieces.

1. Animations (Animations)
2. Cars (Cars)
3. Drones (Drones)

3. Camera & User Controls

This section was broken up into 4 pieces.

1. Global State (Global State)
2. Menu & Overlay (Menu & Overlay)
3. Top down camera (Top Down Camera)
4. Free camera (Free Camera)

Buildings

Buildings are represented by an object model and a list of transformations.

It is assumed that the transformation for each building will remain relatively static and as such all of the transformations for each building shape is stored together (see `src/Building.cppm`).

The buildings are drawn using the `ObjectPipe` pipelines `draw_objects` function. This function takes in a model along with a list of transforms, sends the model to the GPU once and then redraws that model for each transform in the list (See `src/ObjectPipe.cppm`).

```
frame.draw_objects(viewproj, portal.locations, portal.model);
frame.draw_objects(viewproj, hole.locations, hole.model);
frame.draw_objects(viewproj, building1.locations,
building1.model);
frame.draw_objects(viewproj, building2.locations,
building2.model);
frame.draw_objects(viewproj, building3.locations,
building3.model);
```

From `src/cos3712.cpp` Lines 204-208.

The model for each building is loaded when the building is initialized. Locations are later added by calling the function `add_instance`. `add_instance` takes an angle that the model will be rotated by (around the UP axis) and the location that the building will be drawn at (It is assumed that the building is near the origin of the model).

There are a total of 8 buildings that is drawn in the world.

```
portal.add_instance(225.0f, glm::vec3{50.0f, 50.0f, 0.0f});
portal.add_instance(90.0f, glm::vec3{-70.0f, -100.0f, 5.0f});

hole.add_instance(0.0f, glm::vec3{75.0f, -38.2f, 0.0f});
hole.add_instance(0.0f, glm::vec3{-75.0f, 38.3f, 0.0f});

building2.add_instance(0.0f, glm::vec3{-10.0f, 30.0f, 5.0f});
building2.add_instance(180.0f, glm::vec3{10.0f, 40.0f, 1.0f});
```

```
building1.add_instance(0.0f, glm::vec3{-50.0f, 0.0f, 0.0f});


building3.add_instance(0.0f, glm::vec3{50.0f, 0.0f, 2.0f});
```

From `src/cos3712.cpp` Lines 130-143.

Floor

Most of this discussion will concern `shaders/floor.vert`.

The floor is a procedurally generated surface and is drawn centered around the camera (if the camera moves laterally the floor will move with the camera).

The algorithm at its core is very simple.

Construct a grid in row major order (indices increase from left to right, then top to bottom) with N rows and columns.

0	**1**	**2**	**...**	**N - 1**
N	N + 1	N + 2	...	2N - 1
...	...	...	...	...
(N - 1)N	(N - 1)N + 1	(N - 1)N + 2	...	N^2 - 1

Then to calculate the row `row = index % N` and column `col = index / N` is easy.

```
uint Col = gridIndex / NumCols;
uint Row = gridIndex % NumCols;
```

Lines 36-37.

However, each cell in the grid represents a square and a square can be constructed from two triangles. Two triangles has 6 vertices, therefore the `gridIndex = gl_VertexIndex / 6;` can be calculated from the vertex index.

From there the position for the current square is calculated.

```
vec2 xy = vec2(2 * Col * Step, 2 * Row * Step) + pos;
```

Line 23. Where pos is (almost) the position of the camera.

An offset is then added to this position depending on which vertex is being drawn.

```
xy += offsets[square];
```

Line 24. Where square is the index of the vertex (0-5) for the current square.

These values are then used to calculate perlin noise for z value of the final vertex position. The color for each vertex cycles between red, green and blue.

Finally the accuracy for the camera position is clipped so that the floor only updates on whole number intervals of the grid size.

```
template<float step>
static float step_pos(float a) {
    constexpr static float modifier = 2.0f * step;
    a /= modifier;
    return std::floor(a) * modifier;
}
```

From `src/Floor.cppm` Lines 62-67.

```
push_constant_t push_const{
    transform,
    glm::vec2 {
        step_pos<0.5f>(global::cameraPosition().x),
        step_pos<0.5f>(global::cameraPosition().y)
    }
};
```

From `src/Floor.cppm` Lines 92-98.

Animations

Animations are concieved as some sort of function which can update a transform based on a previous transform the total time for which the program has been running and the time difference between the previous update and this one.

The `src/Animation.cppm` module defines two basic animations

1. Translate at a constant rate.
2. Rotate at a constant rate.

As well as various higher order animation constrcuts such as

1. A combination of several animations.
2. A limit to the duration of an animation.
3. A sequence of animations each performed one after the other.
4. A loop of animations which behaves similarly to a sequence except that the animation is reset to a known state before repeating the animation.

There are overloads to binary operators to more easily compose animations.

1. A logical and `&` of two or more animations creates a combination.
2. A logical and `&` of an animation and a duration creates a limited time animation.
3. A logical or `|` of two or more time limited animations creates a sequence of animations.

Cars

Or vehicles.

A vehicle is represented as a shared object model along with a transform. The object is shared to prevent the need for keeping more than one copy of the same object model in memory. Each vehicle has their own transform, because the animation for even similar models can be very different and different vehicles may be using the same model from completely different scopes.

There is currently a single model for a vehicle (`objects/ship.blend`).

There are 5 vehicles drawn and animated in the application.

```
Vehicle ship1;
ship1_animation_t ship1_animation;
Vehicle ship2;
ship2_animation_t ship2_animation;
Vehicle ship3;
ship3_animation_t ship3_animation;
Vehicle ship4;
ship4_animation_t ship4_animation;
Vehicle ship5;
ship5_animation_t ship5_animation;
```

From `src/cos3712.cpp` Lines 97-106.

```
if (global::AnimateCars()) {
    ship1.transform = evaluate_animations(ship1.transform,
total_time - vehicle_stop_time, frame_time, ship1_animation);
    ship2.transform = evaluate_animations(ship2.transform,
total_time - vehicle_stop_time, frame_time, ship2_animation);
    ship3.transform = evaluate_animations(ship3.transform,
total_time - vehicle_stop_time, frame_time, ship3_animation);
    ship4.transform = evaluate_animations(ship4.transform,
total_time - vehicle_stop_time, frame_time, ship4_animation);
    ship5.transform = evaluate_animations(ship5.transform,
total_time - vehicle_stop_time, frame_time, ship5_animation);
```

```
}
else {
    vehicle_stop_time += frame_time;
}

frame.draw_objects(viewproj,
    {
        ship1.transform,
        ship2.transform,
        ship3.transform,
        ship4.transform,
        ship5.transform
    }, *ship1.model);
```

From `src/cos3712.cpp` Lines 180-198.

It should be noted that `vehicle_stop_time` increases as long as the vehicle animations is paused and is subtracted from the total time. If this is not done the implementation of loops breaks for a single loop as the wrong sequence of animations is executed.

Drones

Drones are represented similarly to buildings with a model and a list of transforms. This is because all of the drones has the same animation (a rotation).

There is one drone model `objects/drone.blend`.

```
DroneList drones;
```

From `src/cos3712.cpp` Line 107.

It is drawn 4 times.

```
drones.add_drone(glm::vec3{20.0f, 20.0f, 100.0f});
drones.add_drone(glm::vec3{-20.0f, -20.0f, 100.0f});
drones.add_drone(glm::vec3{-20.0f, 20.0f, 100.0f});
drones.add_drone(glm::vec3{20.0f, -20.0f, 100.0f});
```

From `src/cos3712.cpp` Line 146-149.

The animations will skip the first 2 drones in the list when the option is toggled.

```
void animate() {
    auto update = evaluate_animations(glm::identity<glm::mat4>(),
rotation);
    if (global::AnimateFirst2Drones()) {
        for (auto & transform : locations) {
            transform *= update;
        }
    }
    else {
        for (auto & transform : locations |
std::ranges::views::drop(2)) {
            transform *= update;
        }
```

```
    }
}
```

Global State

There are some actions which can be initiated either from a menu or as a key event. And there is some actions which has repercussions for multiple other systems.

For these cases the appropriate functions to manipulate and access limited shared state is exposed from the `src/GlobalState.cppm` module;

This includes:

1. `windowRef` which required both to display to the screen as well as recieve input from the keyboard and mouse.
2. `CurrentCameraMode` which can be updated from both a menu action and a key event.
3. `cameraPosition` which required to update the camera as well as draw the scene.
4. `AnimateFirst2Drones` which has to be updated from user input and used when animating the drones.
5. `AnimateCars` which has to be updated from user input and used to decide wether to animate any cars.

There is also a utility function `toggleKey` which only triggers an action again if the key was released between presses.

Menu & Overlay

The menu is provided to control various other systems of the applications. However, all systems except for the overlay system can also be controlled using the keyboard and mouse.

The overlay system provides useful information to the user:

1. The controls of the application
2. The camera mode
3. The frame rate

Top Down Camera

In this mode the orientation of the camera is locked such that there is only 4-DOF.

1. Move Forward and Backwards.

2. Move Left and Right.

3. Move Up and Down.

4. Rotate Left and Right.

```
glm::vec3 up = glm::normalize(glm::vec3{orientation.x,
orientation.y, 0.0f});

        global::cameraPosition() += camera_speed * elapsed_seconds
* updateCameraPosition(up);

#pragma region rotation

        if (window.isKeyPressed(GLFW_KEY_Q)) {
            pan_angle += elapsed_seconds;
        }
        if (window.isKeyPressed(GLFW_KEY_E)) {
            pan_angle -= elapsed_seconds;
        }
        orientation = calculateCameraOrientation(pan_angle,
tilt_angle);

#pragma endregion

        return glm::lookAt(global::cameraPosition(),
global::cameraPosition() + glm::vec3{0.0f, 0.0f, -1.0f}, up);
```

From `src/Camera.cppm` Lines 105-121.

Free Camera

The free camera is the oposite of the Top Down Camera where the only restriction is that the camera will always remain "upright".

```
global::cameraPosition() += camera_speed * elapsed_seconds *
updateCameraPosition(orientation);

#pragma region rotation


        static auto PrevMode = global::CurrentCameraMode();
        static auto [prev_x, prev_y] = window.getCursorPos();
        auto [mouse_x, mouse_y] = window.getCursorPos();
        if (PrevMode == global::CurrentCameraMode()) {
            pan_angle -= elapsed_seconds * static_cast<float>
(mouse_x - prev_x);
            tilt_angle -= elapsed_seconds * static_cast<float>
(mouse_y - prev_y);

            orientation = calculateCameraOrientation(pan_angle,
tilt_angle);

        }
        PrevMode = global::CurrentCameraMode();
        prev_x = mouse_x;
        prev_y = mouse_y;

#pragma endregion
        return glm::lookAt(global::cameraPosition(),
global::cameraPosition() + orientation, glm::vec3{0.0f, 0.0f,
1.0f});
```

From `src/Camera.cppm` Lines

Assignment 4

This section is about the features which was implemented for assignment 4.

Lights

1. A sun was implemented as a directional light which orbits the scene. The colour and position of the sun can be changed through the `Lights` menu in the overlay.

Shading techniques

1. The flat shading technique was implemented.

Textures

1. The floor has a color texture.