Graphics phase 3 olive

1. Skim over the project's code that you havent looked at before (play-state.hpp, component-deserializer, etc..). Try to focus on understanding entities, components, and systems since those are where the game logic is written and executed.

2. Open the application with `-c='config/app.jsonc'` . This is a scene that has almost everything from phase 2 and should be a good starting off point for you to build off. Create copies of it and modify them depending on your use case.

3. See how states are added in `main.cpp` . This should tell you how to create your own stat with custom systems.

4. Try to add and deserialize a simple component.

   i. Create a new .hpp file containing the component's data. You can look at `source/common/components/movement.(hpp/cpp)` as an example. Recall that components should contain at most data and a few helper methods. The bulk of the logic should be in systems operating on that component. Don't forget to add a `deserialize` overload to read the component data from the scene json.

   ii. Create a corresponding system that operates on this component, usually the system has an `update()` method similar to the following:
   Note that the name of the method `update()` is up to you since you'll be calling it in the state manually later on.

```
void update(World* world, ...parameters that change every frame) {
      // For each entity in the world
      for(auto entity : world->getEntities()){
          // Get the movement component if it exists
          MyComponent* myComp = entity->getComponent<MyComponent>();

          // If the component exists
          if(myComp){
              // read data from myComp
              // operate on it
          }
      }
  }
```

iii. Create a new scene based on `play-state.hpp` and add it in the main function. Note that the field `"start-scene": "scene name"` should match `app.registerState<MyState>("scene name");` Your `onInitialize()` method should look something like this:

```cpp
void onInitialize() override {
   // First of all, we get the scene configuration
   // from the app config
   auto& config = getApp()->getConfig()["scene"];
   // If we have assets in the scene config, we deserialize them
   if(config.contains("assets")) {
       our::deserializeAllAssets(config["assets"]);
   }
   // If we have a world in the scene config,
   // we use it to populate our world
   if(config.contains("world")) {
       world.deserialize(config["world"]);
   }

   // Scene specific initialization logic
   // Example: one unique entity that you want a pointer to
   // that won't change during gameplay
}
```

Your `onDraw()` method should look like this:

```cpp
void onDraw(double deltaTime) override {
   system1.update(param1, param2, deltaTime);

   // world is stored in the state and initialized in onInitialize()
   collisionSystem.collide(world);

   // Other systems
}
```

Your `OnDestroy()` method should look like this:

```cpp
void onDestroy() override {
    // Don't forget to destroy the renderer
    renderer.destroy();

    // On exit, we call exit for the camera controller
    // system to make sure that the mouse is unlocked
    cameraController.exit();

    // and we delete all the loaded assets to free memory
    // on the RAM and the VRAM
    our::clearAllAssets();
    // De-initialization logic for other systems
  }
```

Optionally, you can overload `onImmediateGui()` to display a GUI for debugging/gameplay purposes:

```
  void onImmediateGui() override {
    ImGui::Begin("KAK Engine"); // Must begin before ANY ImGui functions

    ImGui::Text("Some important data: %s", importantDataString);
    if(ImGui::Button("Button name")){
      // Do something on press!
    }

    ImGui::End(); // Must end before the next `ImGui::Begin()` call.

    // Can have multiple windows too
    ImGui::Begin("Debugger");

    auto entities = world->getEntities();
    for(int i = 0; i < entities->size(); i++){

      auto entity = entities[i];

      if(auto buggyComp = entity->getComponent<BuggyComponent>()){
        // %d for integers, corresponding argument MUST be an integer.
        // %f for floats, ...
        ImGui::Text(
          "Buggy component %d floatField: %f",
          i, buggyComp->floatField
        );

        // or

        // %s for strings, you can create a `toString()`
        //method in your component to convert its data to
        // a string for printing
        ImGui::Text(
          "Buggy component %d: %s",
          i,
          buggyComponent.toString()
        );
      }
    }

    ImGui::End();
  }
```