Phase 2 Help Document

In this document, we will break down the requirements of phase 2 to make it more clear.

Part 1: Game State Management

In the examples, we have a class “Application” that handles the game loop. During the application lifetime, we have 3 phases:

1. Initialization (onInitialize): which is executed once before the game loop.
2. Render Loop: while not exiting, execute:
    a. Drawing (onDraw): where we draw the scene.
3. Destroy (onDestroy): which is executed once after we exit the game loop.

This works great for the examples but for a game, this has an obvious drawback. This means that we must do all the initialization once and all the destruction once. This is unreasonable for almost all games where we have multiple states. Example of these states are:

- Menu state: where we have a UI but no logic for the game. So it is unreasonable to load all the models, shaders and textures that are used in gameplay. Also if we use a physics engine, there is no reason to initialize a physics world for a menu. All of these are a waste of resources.
- Play state: where the player actually plays. Where we need resources and objects related to gameplay but we don’t need any resources or objects related to the menu.

So we need to refactor our structure a little bit. Instead of extending our application to implement our custom logic, we will convert it to a state manager and implement all of our logic in what is called a game state. The base class for game states should have at least these 3 functions:

1. “onEnter”: which is called once before we use the state.
2. “onDraw”: which is called once every frame to draw the scene.
3. “onExit”: which is called once before we release the state.

So now the application should have 2 pointers: a pointer to the current state and a pointer to the next state. Initially, both pointers will point to null. we should add a function “goToState” which is given a state pointer that will be assigned to the next state pointer. The application can now be changed to work as follows:

1. While not exiting, do:
    a. If the next state is not null:
        i. If the current state is not null, call “onExit” of the current state.
        ii. Set the current state = the next state.
        iii. Set the next state = null.
        iv. Call “onEnter” of the current state.
    b. If the current state is not null:
        i. Call “onDraw” of the current state.
2. After the loop, if the current state is not null, call “onExit” of the current state.

This would ensure that for any state, “onEnter” is always called once before any “onDraw” and “onExit” is called once after we finish using the state.
Now, we can extend the game state to create our menu state and play state. Before we start running the application, we should call “goToState” to give it the initial state of the application. For example, if the game starts with the menu state, call “goToState” and give it a pointer to a menu state before calling run.

Note that I omitted any mention of “onImmediateGUI” for simplicity. If you plan to support Immediate GUI in your game, add a function for it in the game state and call it in the application.

You can also split or add functions as you wish to the game state to fit your needs. For example, you can add callbacks for key and mouse events and call them from application.

Part 2: ECS

There are many ways to implement an ECS so I will discuss 2 approaches.

The first approach uses data oriented design. This is what you’ll find in most ECS libraries online. It is great for performance, cache friendly and usually supports multithreaded systems. However, it is harder to implement and can feel restrictive if the developer is not yet comfortable with ECS. For this implementation of ECS, you can define the framework as follows:

- Entity: It will be just an integer containing the id of the entity.
- Component: It can be any “struct”. It should contain only data. No need for a base component type since any “struct” can be a component.
- Component Manager: It contains an array of components of the same alongside an array of entities (integers) holding each component. Usually we assume that each entity can only own a single component of each type. You can have a manager of each component type or create a manager for all the components.
- System: It defines the logic of a specific part of the game. It can request to access entities that hold a certain combination of components.
- World: just a container of entities (integers).

The second approach is much more relaxed since it will not be as performance oriented as the data oriented architecture of ecs. In this way, you can define the framework as follows:

- Entity: it is a class that contains a container of components. The container can be a map, a set, a list, etc. It should have some basic manipulation and access functions such as “getComponent”, “addComponent”, “removeComponent”, etc. No game logic should be defined in the entity class.
- Component: you can define a base class for all components and have all components extend it. Since we have already lost the benefit from restricting the components to have data only (cache friendliness), you may break the rule and add logic to the components if you wish. Overall, you don’t need to add anything special to the base component class and you can keep it empty if you wish.

- System: It defines the logic of a specific part of the game. It can be a class, a function, or whatever you may find convenient.
- World: just a container of entities (integers). Use lists, sets, vectors or whatever you like.

Pick whatever approach you like. There are still many design decisions that are left for you to make regardless of which approach you pick.

Part 3: The Transform components.

The transform component defines where the entity is in the 3D space. It should have at least these 4 fields:

1. Position: 3D Vector representing the translation of the entity.
2. Rotation: it can be Euler Angles, Axis-Angle or Quaternions. It represents the orientation of the entity.
3. Scale: 3D Vector representing the scale of the entity.
4. Parent: It can be a pointer to its parent entity or the transform component of it's parent entity. For data oriented ECS, it can be the id (an integer) of the parent entity. This is used to define the scene graph.

You can add any other data you may find convenient to the component such as a list of children.

Part 4: The Renderer and MeshRenderer components

For the Renderer component, it should be a base component for all renderer components. The MeshRenderer should extend it and be responsible for rendering a single mesh to the scene. So it should contain the following information:

1. A pointer to the mesh to be rendered.
2. A pointer to the shader that it should use for rendering.
3. Any data that should be set as uniforms on the shader (except the transformation data which should be retrieved from the transform component).

It is common to wrap point 2 and point 3 into a single class called "Material" and have a pointer to a material object in the mesh renderer component. This will be done in Phase 3 but you can do it in this phase if you wish.

Note that when using a data-oriented ECS architecture, it is meaningless to have inheritance since components shouldn't have any logic. So if you go for this architecture, you can skip the Renderer component and write the Mesh Renderer Component only.

Part 5: The Camera component

This should contain all information about the camera projection such as the projection type (orthographic or perspective), the field of view angle, the near and far planes, etc. Note that we didn't mention the camera view transformation since this information should be retrieved from the transform component.

Part 6: The Mesh class

This class should hold an OpenGL “vertex array” and all the accompanying vertex and element buffers. It should be responsible for creating, destroying, populating and drawing these objects. See the Mesh class in the examples for more details.

Part 7: The RenderSystem class

This class is responsible for rendering all the renderer components in the scene. The most basic implementation of a RenderSystem is a function that loops over all the entities and for each entity that has a transform component and a renderer component, it will do the necessary steps to render it.

The rendering logic can either be implemented in the renderer component or inside the render system itself.

Part 8: Shaders that are compatible with the RenderSystem and Renderer components

You almost have nothing to do here since the shaders are already written in the examples and you only need to merge it with the project. See if you want to make it feel comfortable for you (add your own comments, modify the variable names as you like, add any features you may want, etc).

Part 9: CameraController component (and optionally a System)

Now we want to write a component that would allow you to control the camera via keyboard and mouse. It should be added to the entity that has the camera and it should move the transform of that entity. See the “FlyCameraController” class in the examples which has the logic for controlling the camera but was not designed to work in an ECS framework. Your task is to make it fit in with your framework.

That logic may be implemented in the camera controller component or in a system that finds the entity with the controller component and controls its transform.

Part 10: An Example Scene

In this part you should create a game state that does the following:

1. Create an entity with a transform component, a camera component and a camera controller component.
2. Create at least 2 entities: each having a transform and a mesh renderer component. Each entity should have a different position and a different mesh.

The state should have all the necessary code to initialize the resources, create the systems, entities & world and call the functions necessary such that we can see the entities rendered on

the screen and can control the camera. Don’t forget to also release all the resources while exiting the application.