Computer Department
Faculty of Engineering
Cairo University

CMPN205
Computer Graphics and
Man-Machine Interfacing
Fall 2020

Project Description

The learning outcomes of the project are:

1. Learn how to build a rendering engine using a modern graphics API (OpenGL 3.3+).
2. Implement a popular game engine design paradigm which integrates the rendering engine as an isolated module.
3. Use the built engine to create a 3D game.

Introduction

A Game Engine combines modules with distinct functionalities (e.g. Graphics, Audio, Networking, Physics, User Input, AI, Asset Management and Game Logic). Within each module, there are lots of complexities that usually require experts from different backgrounds working together on the same product. On top of that, game engines have to process frames in a matter of a few milliseconds (~16 ms for 60fps), so efficiency is a top priority. Additionally, it has to manage a satisfactory compromise between flexibility and abstraction such that it can support the creative and complex process of game development without requiring long iteration cycles. To some degree, the same requirements apply to other graphics applications.

To support these requirements, game engine architectures have developed over decades to keep up with the ever-rising complexities of video games. Currently, game engines have become so flexible and advanced that they have been employed in other industries such as movies, virtual training, architectural visualization and much more. Most of the modern engines, such as Unity, Unreal Engine and CryEngine, adopt an Entity-Component (EC) architecture. We will discuss it alongside a close relative called Entity-Component-System (ECS).

For the project, we will implement an ECS framework with the components and systems required to abstract the rendering logic from the rest of the game logic.

Project Architecture

Game States

It is common to have multiple distinct states in any game. For example, a game may have these states:

- A menu state where no game logic is running but a user interface is used to select options and start a game.
- A play state where the game logic is running.

While it is possible to write code for both states in the same class, it can get messy pretty quickly and the extensibility will only suffer as new states are added. To keep our code manageable, we will divide them into 2 parts:

1. A game state manager which is responsible for calling state methods, switching between states and so on.
2. A game state which is a class that will be the base of all the game states. At the very least, a game state should have 3 virtual functions:
    a. "On Enter" which is called as soon as the state becomes the current state. It should handle any state-specific initialization, resource loading, etc.
    b. "On Draw" which is called every frame to draw the state. It can also handle updating the state.
    c. "On Exit" which is called as soon as the state is no longer the current state (or the application is exiting). It should free any loaded resources.

The game state manager should store a pointer to the current state. In the state logic, we should be able to request a transition to another state. However, the game state manager shouldn't replace the current state as soon as the request is issued; it should wait till the current frame ends. One way to implement this is to store a pointer to the next state which should be null by default. At the start (or end) of every frame, the manager will check if the next state pointer is not null (a transition request has been issued). If yes, it should call "On Exit" on the current state, replace it with the next state, call "On Enter" on the new current state then make the next state pointer null again.

Note that the details we mentioned above can be modified as you see fit. For example, it is common to split the function "On Draw" to 2 functions: "On Update" for updating the state and "On Draw" for drawing the state only. We will leave such design decisions for you to make.

RAII (Resource Acquisition is Initialization)

This is a very popular idiom which denotes linking the resource acquisition and release to the lifetime of an object. For example, if we encapsulate an OpenGL shader program in a class using RAII, we would write it as follow:

- In the class, we will store the name of the program in an unsigned integer.

- During construction, we could create the program.
- Most importantly, during deconstruction, we could delete the program.

However, care must be taken when it comes to copy constructors & assignment operators. We cannot naively copy a resource. For example, if we copy an object that encapsulates a shader program using the default constructor, we would end up with 2 objects holding the same name (unsigned integer). If any of them is deconstructed, it will delete the shader program and the other object won't notice thus we could end up with runtime errors. To solve this, we have 2 approaches:

1. Delete the default copy constructor and assignment operator.
2. Override the copy constructor and assignment operator such that they create a separate resource.

Entity Component System (ECS)

We will briefly describe what an Entity-Component-System is. For a background about "why to even use ECS" is in the appendix.
ECS is a very popular design pattern for separation of concerns and it stands for Entity-Component-System which represents the 3 main parts of the pattern. In games that utilize ECS, each object could be seen as an entity. The entity is just a container of components. It usually holds no logic. For example, a player character could be seen as an entity. If we take a look at all the features needed for a player character, we can split each of them into a separate component:

- It will need to hold a component responsible for its visual aspect (Renderer Component).
- It will need to store its location and orientation in the world (Transform Component).
- It will need to define its physical shape (Physics Component).
- It will need to store or receive the user input (Input Component).
- And so on...

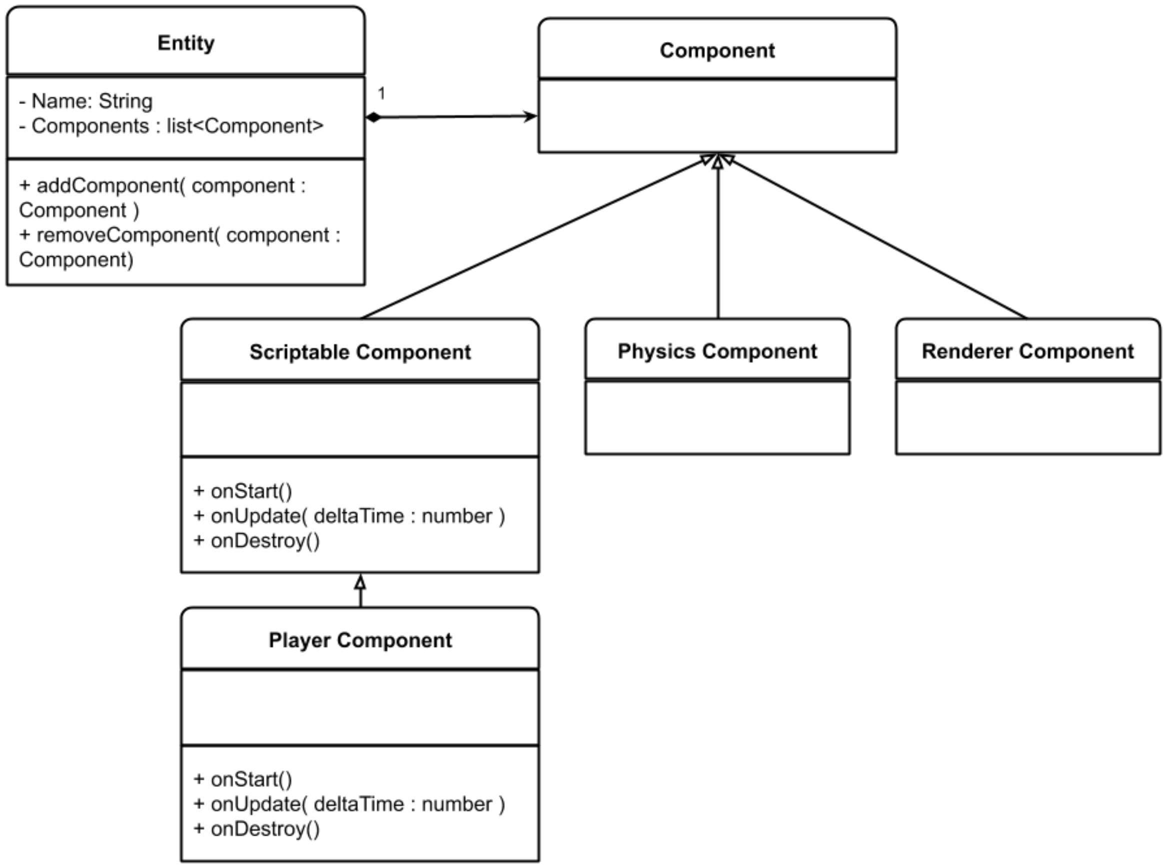
It is the responsibility of the engine designer to decide the components and how much the responsibilities should be separated or combined. For example, the transform component usually holds the position and the orientation. In some games, they are split into a position component and a rotation component.

In the Entity-Component (EC) framework, logic is usually handled by the components. In ECS, components usually hold data only and all the logic is done by systems. For example, a Render System is almost always needed to render entities holding a renderer component. In that case, the renderer component is mainly used to keep the data about how it should be rendered and it almost has no rendering logic in it. Although the project will require an ECS framework, we can break the rule of keeping logic out of components for the sake of convenience.

Finally, we usually need to group entities together so we can create a "World" to hold a collection of entities. Now systems can refer to a world object to search for entities that they are concerned with.

To sum up, An ECS framework can be divided into 3 parts:

- **Entity**: It should hold a list of components. While formally entities do nothing else, we can add some pieces of logic to it (e.g. start, update or destroy components).
- **Component**: It can hold data and/or logic. A component should only be owned by a single entity. An empty component which has no data or logic is usually named a “Tag Component” since they can be used to tag entities.
- **System**: It can handle the logic of the world. It can read/update entities that hold certain components. Some systems can hold data which is global to the system and not specific to a certain entity. Systems can be as simple as a single function or be a complex class. That’s why we didn’t include it in the example UML diagram.


Entity
- Name: String
- Components : list<Component>
+ addComponent(component : Component)
+ removeComponent(component : Component)
1
Component
Scriptable Component
+ onStart()
+ onUpdate(deltaTime : number)
+ onDestroy()
Physics Component
Renderer Component
Player Component
+ onStart()
+ onUpdate(deltaTime : number)
+ onDestroy()

Requirements

The requirement is divided into 2 parts: the engine and the game. The game code will use the engine code.

The engine must include the following features:

1. Game state management.
2. An ECS framework.
3. A Graphics Abstraction Layer over OpenGL using the ECS framework. In other words, the game code should not need to directly call any OpenGL functions.
4. Resource-Holding Classes (such as meshes, textures, etc) must be compatible with the RAII idiom.
5. Deserialization for loading scene and resources from files.

The game must have the following:

1. Use all of the aforementioned engine features.
2. Must have some form of game logic where the player has to make decisions and take actions to achieve a goal.
3. Have at least 2 Game States with different purposes (e.g. menu state and play state).
4. All scenes and paths to external resources (model, textures, etc) should be defined in external non-code files. No paths or scene data should be hard-corded except the paths to the scene and resource definition files.
5. At least one 3D model must be loaded from an external file (either created using any modeling program or downloaded from the internet).
6. Must include lights. The shaders should be able to support multiple Lights that can simultaneously affect the same object.
7. At least one model must have textures applied to it.
8. The objects must be movable by the player on 2 or 3 axes.
9. Must include any form of collision detection in 3D (obstacles, ray-picking, etc).

The code must be cleanly structured and well organized.

Delivery Phases

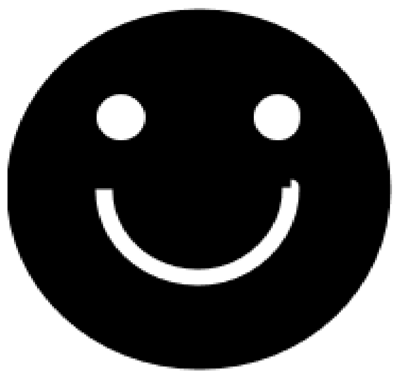
Phase 1 (Due Date: Week 4):

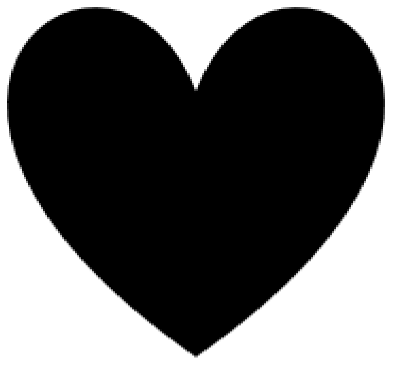
Engine:

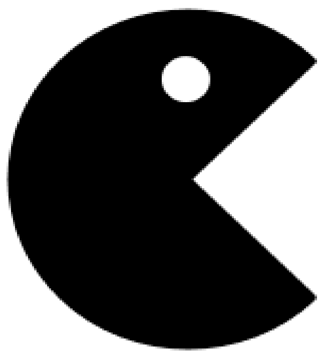
- Create the project structure.
- Add Window management and user input.
- Write a shader class that handles an OpenGL program (RAII Complaint).

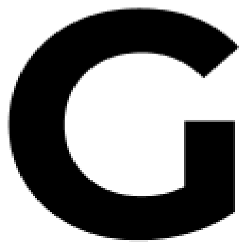
Secondary:

- Write a vertex shader that covers the whole screen.
- Each member of the team should select one shape of the following (no two students should select the same shape) and write a fragment shader that approximately looks like the selected shape. The shape should follow the mouse pointer.


Shape 1


Shape 2


Shape 3


Shape 4

- Load the shaders.
- Using the keys 1 to 4 to select a shader and draw it.

Phase 2 (Week 7):

Engine:

- Implement Game State Management.
- Write the Entity, Component and World classes. World is a container of Entities.
- Write the Transform component (contains Translation, Rotation, Scale and Parent).
- Write the Renderer & MeshRenderer components.
- Write the Camera component.
- Write the Mesh class (RAII Complaint).
- Write the Render System.
- Write the shaders that support the defined system (contains 3D transformation).

Secondary:

- Write a component (and a system if needed) that moves the camera using the mouse and the keyboard. Use WASD to move forward, left, backward, right. Use QE to move up and down. Use the mouse to rotate the camera along its X and Y Axes.
- Create a scene with at least 2 visually distinct objects and a camera with its controller.

Phase 3 (Week 10):

Documents:

- Game Proposal Document.

Engine:

- Create texture2D and Sampler class (RAII Complaint).
- Create a RenderState and Material class.
- Modify the RenderSystem to support materials and render states.
- Write the shaders that support the defined system (supports textures).
- Add Scene Deserialization.

Secondary:

- Create a scene that has textures.

Final Delivery (Week 12):

Report:

- Same as the proposal but with at least 3 actual screenshots from the finished game.
- Work distribution across the team members.

Engine:

- Create a Light component.
- Modify materials to support texture maps for lighting.
- Modify the RenderSystem (to Support Lights and Ambient Light)

Game:

- Implement Game Logic.
- Finish the rest of the requirements.

Deliverables

Game Proposal Document (Phase 3): It should contain a game name, a paragraph describing the game and an image demonstrating the game idea. The description should describe how the game will be played and what is the goal of the game. Don't forget to add the team member names, IDs and team number. Some example proposals are included in the appendix.

Final Report (Phase 4): Same as the game proposal but should include at least 3 screenshots from the finished game. It also should include the work distribution on the team members.

Project Code (Phases 1-4): The delivered project should include everything needed to build and run the game. It also should include a brief explanation of how to build the game. It must include a file with the team number, member names and IDs.

Deadlines

The deadline for each phase will be the start of the tutorial period in which the team is registered in the project teams sheet.

Grading Notes: Please note that the main focus of the project is to apply graphics algorithms. The focus is not about art, game design, audio, etc. Don't put too much effort in things that deviate from the main goal of the project. Each student will be graded individually according to their contribution and understanding of the project. Each student is expected to understand all the covered topics about the rendering process using OpenGL.

Appendix

Entity Component System Background

First, let’s talk about one of the most popular programming paradigms: Object Oriented Programming (OOP). Besides being popular, it is also very controversial. Since we are not here to defend or criticize OOP, we will just show one example where naively using inheritance can lead to bad software design especially for games.

Imagine we are making a fantasy game in a land filled with mages, goblins and elves. So let’s start by creating our base class class for all characters and call it “Character”. The class “Character” will have virtual functions for rendering the characters, physically interacting with the world, generating sounds, attacking, handling health, etc. Now let’s extend it and build our class “Human”. Humans can have different professions so let’s extend it to “Knights”, “Mages” and so on. Now we need to add elves so let’s have an “Elf” class. Elves also have professions so let’s extend “Elf” to create “Knights”, “Mages” and…. Wait a second. We already have Knights and Mages for Humans, so these will be “ElfKnights” and “ElfMages” which will be totally separate classes. Now we have lost a blessing of Inheritance which is “Reusability” since we need to rewrite any common parts of Knights and Mages twice; once for humans and once for elves. Let’s change our design a bit.

Lets separate Humans from their professions. So Knight and Mage will no longer extend human; they are generic professions now. This way we can have a human knight which extends 2 classes: Human and Knight. Now we can combine races and professions as we want and avoid repeating our code. But wait… both “Human” and “Elf” extend “Character”, but what does “Knight” and “Mage” extend. We can choose not to extend anything but we will lose the advantages of polymorphism. Or we can make a base class for professions (e.g. “Profession” class). But what if the professions need to modify the character behaviour (e.g. Knights may be able to recover stamina over time which is a character behaviour), we would need to extend “Character” itself. If we allow professions to extend characters, we fall into the trap of the “Deadly Diamond of Death” where a class extends 2 or more classes who have a common base class. So let’s scrap all of this and start cleanly without relying on inheritance.

Let us introduce the principle of “Composition over Inheritance”. Simply put, this principle says that we should achieve code reusability using composition rather than inheritance. This cleanly fits with our requirements. Instead of creating a human mage via inheritance, we can make a generic game object and add two components to it: a human and a mage. Now both components can access each other via their parent game object and thus they can affect each other. Now the professions no longer need to extend “Character”, they can interact and affect the behaviour of the other components in the same game object.

Now let’s go slightly overboard and further split components down to their basic responsibilities. For example, let’s make separate components for rendering, physics, audio, health, attacks, etc.

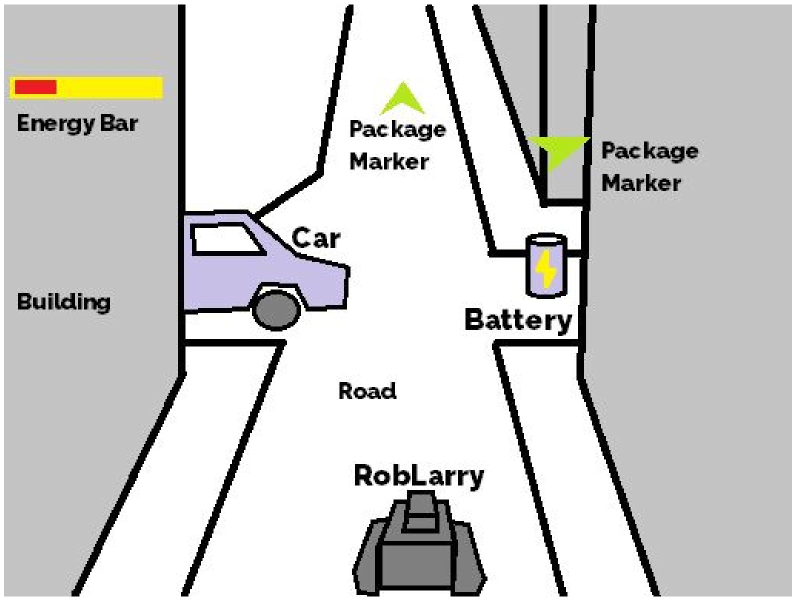
Now the game object itself can be just a container that holds these components; in other words, it can be seen as a data structure (list, set, vector, etc) that can contain any combination of components we would like. Now we have what is called the Entity-Component (EC) framework which is used by popular game engines such as Unity. If we go a bit further by enforcing that component should only contain data and that all the logic is handled by external code called “Systems”, we would have what is called the Entity-Component-System (ECS) which is the latest trend in the game development community. With ECS, we can 100% remove inheritance from our code (and yes, some modern games and engines are written purely in C). However, no need to be so extreme. Since we are not aiming for the performance of a AAA engine, we should design the framework with convenience of use as our top priority as long as the performance is acceptable for a real time application (no less than 30 frames per second).

Proposal Examples

Lamazone Delivery Service

Lamazone online shopping service has built the perfect delivery robot, a robot that can never stops moving. You play as a RoboLarry who is new to the job and has to finish his daily delivery quota. Your battery is limited and you can only hold one package at a time. Go to storage locations to take a package and deliver them to their corresponding customers. Steer left and right to avoid hitting buildings and moving cars and collect batteries on the road to recharge. Beware, the more energy you have, the faster you’ll move.

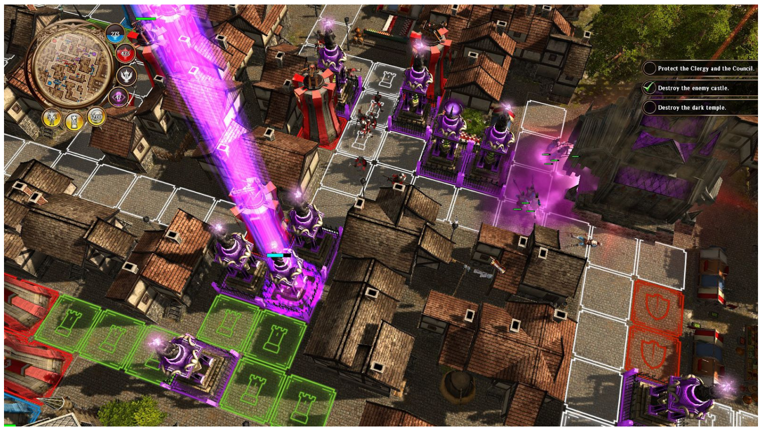
Example Image (Drawn in Paint):



Towers of the Demon Lord

The army of light is approaching the city of the demon lord so he hires you, the underworld engineer, to build him a defense system against them. Add turrets, traps, and other obstacles to stop them from reaching the demon lord’s castle. Collect coins from dead knights to buy more tools and materials.

Example Image (Photo from Defenders of Ardania)



Protect the Clergy and the Council.
Destroy the enemy castle.
Destroy the dark temple.

Nice Day to Die Again

It was a nice day so you went shopping with your friends. Unfortunately, the zombies thought it was a nice day to go shopping too. Use your machine gun to make sure they are dead again. Get all the shopping items you want and get to the cashier without allowing any zombie to bite you. Shoot them if they get near you.

Example Image: (Assembled together in GIMP)

Rice Home Baking Sugar
Milk Powder Liquid Milk
New Arrival
BEST PRICE EVERYDAY
Lilac Cup Cake Vanilla Chocolate Per Box
5.95