n8 Game Framework Design Document

Nate Ebel

March 26, 2014

Contents

1 Current Ideas and Things to Think About

- Using a more “pure” entity-component system with systems operating on components and the “entity” or game object just acts as an id to relate components loosely to one another
- Systems can be treated as services that are globally available through a service locator
- . . . more ideas and full new design for ECS in notes

2 Design and Use Patterns

2.1 Command

2.2 Observer

2.3 Subclass Sandbox

2.4 Singleton

2.5 Service Locator

2.6 Components

3 Client Use/Interaction

Client should instantiate a single instance of a Game object.

3.1 System wide values and enums

3.2 Configuration data/files

4 Engine Core

4.1 Game Loop

Takes place within Game and uses the Game object’s GameTimer instance to control the loop. Within the loop the current GameState is processed at each iteration.

4.2 Window Management

Should be a window object. Should support renaming and resizing.

4.3 State Management

5 Game States

A GameState is whatever is happening in the forefront of the game. GameStates will be maintained on a stack within a GameStateManger. GameStateManager acts as a bridge between the game loop within Game and the currently active GameState. Should support passing of entities between states, and each GameState will maintain a list of its entities. GameStates will include 3 key methods of responding to user input, updating, and rendering. The Update method will receive the current time as an argument, and Render will receive the current window canvas as an argument. These 3 methods will be called in the game loop within the Game object. Pause and resume methods can be overridden to handle changes to and from the GameState.

6 Entity-Component System

The entity-component system will be the core of gameplay, logic, and game customization.

6.1 Entities (GameObjects)

An object visible within the game's screen will be an entity of GameObject. These could include a chest, a power up, or a user controller character. GameObjects should act as a common identifier between components, and as the central storage location for each component. The behavior and attributes of a GameObject will depend solely on which components are related to the GameObject. For example, the difference between a stationary enemy and one that moves around could be the additional of a "movement" component that updates the enemy's position frequently. GameObjects will be created and stored by a GameObjectFactory that will be customized by the client to produced the desired object types. When the GameObjectFactory creates an object, it will register it with the necessary Systems. GameObjects will maintain a list of their associated components. Pointers to these components will be stored by Systems in order to do batch component processing.

6.2 Components

A component will contain the data needed to provide some specific functionality to a GameObject. Every game object will have a "Location" component to track where the object is and its size. Components will not contain methods, and instead will be operated on by "Systems". These systems will store pointers to the components they operate on. When an entity is registered with a System, if the entity possesses the required component a pointer to that component will be stored in the system to enable batch processing. Components should provide registration functions to handle registering and unregistering themselves from Systems.

6.3 Systems

A system provides behavior for, and operates on components. They are responsible for updating components during each iteration of the game loop. A system will store each object that it operates on so they are together in memory when System.Update() is called. When a component is removed from a system, a member variable counting the current components will be decremented, and a placeholder object inserted so other components' handles aren't affected. When a new component is created, if the size of the vector and the count of components aren't the same, the new component will be inserted at the empty position.

Systems will be able to communicate with one another using the Observer pattern. During engine setup, default systems that need to communicate will be registered as observable and subject objects. Game specific systems can register to observe other systems when they are created at game startup.

6.4 Default Systems

6.4.1 Input

Input will be handled by an InputSystem that tracks which keys have been pressed down or are up. These states will be able to hold pointers to Command objects that define an action to take. Game states can then register which commands are associated with which inputs when the state gains focus.

6.4.2 Rendering

- Render single entity

- Render single entity with a camera object (Camera: see Section **??**)

Static sprite Animated sprite

A camera object can be passed in to an entity's render method to have it draw based on the camera's current position.

6.4.3 Audio

6.4.4 ResourceManager

1. Resources types (Image, Texture, Audio) will be subtypes of a base resource class. Each resource type will have a designated directory relative to the application directory where resources will be stored. In this directory will be a file that contains the file names of resources in that directory to load. When the ResourceManager loads resources, it will look in these directories, load the file, load all files and create the necessary resource objects, and store them in a map.

6.5 Game Specific Systems

6.5.1 Physics

6.5.2 Game Logic

6.5.3 AI

6.5.4 UI

6.5.5 ObjectFactory

7 Globally Available Services

Use globally available service locator. The service locator should support registering and unregistering of services, and retrieval of those services. Ensuring that a service is available before be used will be the job of client code. Systems will be stored and served by the service locator. This includes default systems, and game specific systems registered by the client.

7.1 Logging

Logging should be available to any aspect of the system through the ServiceLocator.

8 Game levels

9 Use Cases

9.1 Input changes game state

9.2 Input changes player state

walking → flying

9.3 Input moves player

9.4 Entity destroyed and animation is played

9.5 Timed event activated

9.6 Timed event destroyed

9.7 Entity collision

9.8 AI player must make a decision

9.9 Notify an achievement system when 5 enemies are destroyed

9.10 Remap an action's key