Lecture 2

Game Engine Components

CS230
Game
Implementation
Techniques

1. Game Engine Design

1.1. Overview

- A game engine is divided into 2 components: System component and Game Logic component.
- These two components should be totally separated. One deals strictly with the hardware while the other deals with game logic.
- Each subcomponent should belong entirely to either one of these two components. If a subcomponent turns out to be contributing to both components, it should be divided to 2 or more several subcomponents, where each of them satisfies the aforementioned rule.

1.2. System Components: Introduction

- The system component communicates with the hardware.
- It isolates the game logic component from directly communicating with the hardware.
- This isolation makes it easier to port the game to another platform.
- The system component is broken into several subcomponents: Memory manager, graphics manager, audio manager, frame rate controller...

1.3. Game Logic Components: Introduction

- The game logic component is where the game code resides.
- It does not communicate directly with the hardware.
- It does not even know what the underlying hardware is.
- When the game code needs to communicate with the hardware, it uses system components functionalities in order to do so.
- It also contains the physics/collision engine, object manager, environment manager, messaging system, game state manager, camera system...

- **Managing system components**

  Any game code (or application in general) has to initialize some system components before actually going into the game code and the game loop. These system components usually set up some necessary hardware related functionalities which will be later used within the game. Example:

  - Setting up a input device
  - Setting up video device
  - Setting up an audio device
  - Allocating video buffers
  - Deciding if some parts of the pipeline should be done using the hardware or software.

  This kind of system component initialization should be done just once before entering the game loop, and if any component fails to initialize properly, the application or the game usually quits with the appropriate error, since the application won't be able to run.

If all system components are initialized properly, we initialize few arguments like the frame rate controller and the previous/current/next game state and the game loop is started. Note that the previous code should never be reached again.

Upon exiting the game loop, all the devices that were allocated should be released before exiting the application. For example, if a video device was created, it should be released in order to free all its allocated resources on the video card. Also, if an input device was allocated during the initialization stage, it should be released upon exiting the application. This would be the final code part of the game or the application, and care must be taken to make sure every allocated resource is released.

1.4. System Components

- **Memory manager**
  - The memory manager is responsible for allocating/deallocating memory.
  - This is usually done by the operating system, but it's always better to have a layer between the game engine and the operating system, which in this case is the "Memory Manager".
  - If a memory manager is implemented, the game should deal with it in order to allocate/deallocate data, and not directly with the OS.
  - A memory manager should include these features (In order of importance):
    - Optimizes memory allocation/deallocation. The algorithm should be written in a way to suit the game's style (Block size, frequency...)
    - Hierarchical memory heap. Deallocating a heap will automatically deallocate all its children heaps. Example: Upon deallocating a level's heap, all its children' heaps (the memory allocated for all objects contained in the level) should be automatically deallocated.
    - An easy way to track memory usage. This is done by tagging each allocated memory, and by having a function that reports memory usage at any time.
    - A simple way to track and catch memory leaks. This relies also on the fact that all allocated data are tagged, which makes it easier to determine which allocated data wasn't deallocated.
    - Support for smart pointer.
    - Run-time memory defragmentation. Whenever the CPU is idle, the memory manager should automatically move memory blocks around to minimize memory fragmentation.

- **Graphics manager**
  - As graphics hardware is getting more advanced, the steps required to put just a simple thing on the screen gets more complicated.
  - There are more and more steps involved and more chance to get it wrong and not having anything on the screen. Given the situation, the Graphics Engine's main job, on top of

encapsulating the graphics hardware, is to simplify the process to get something to the screen.

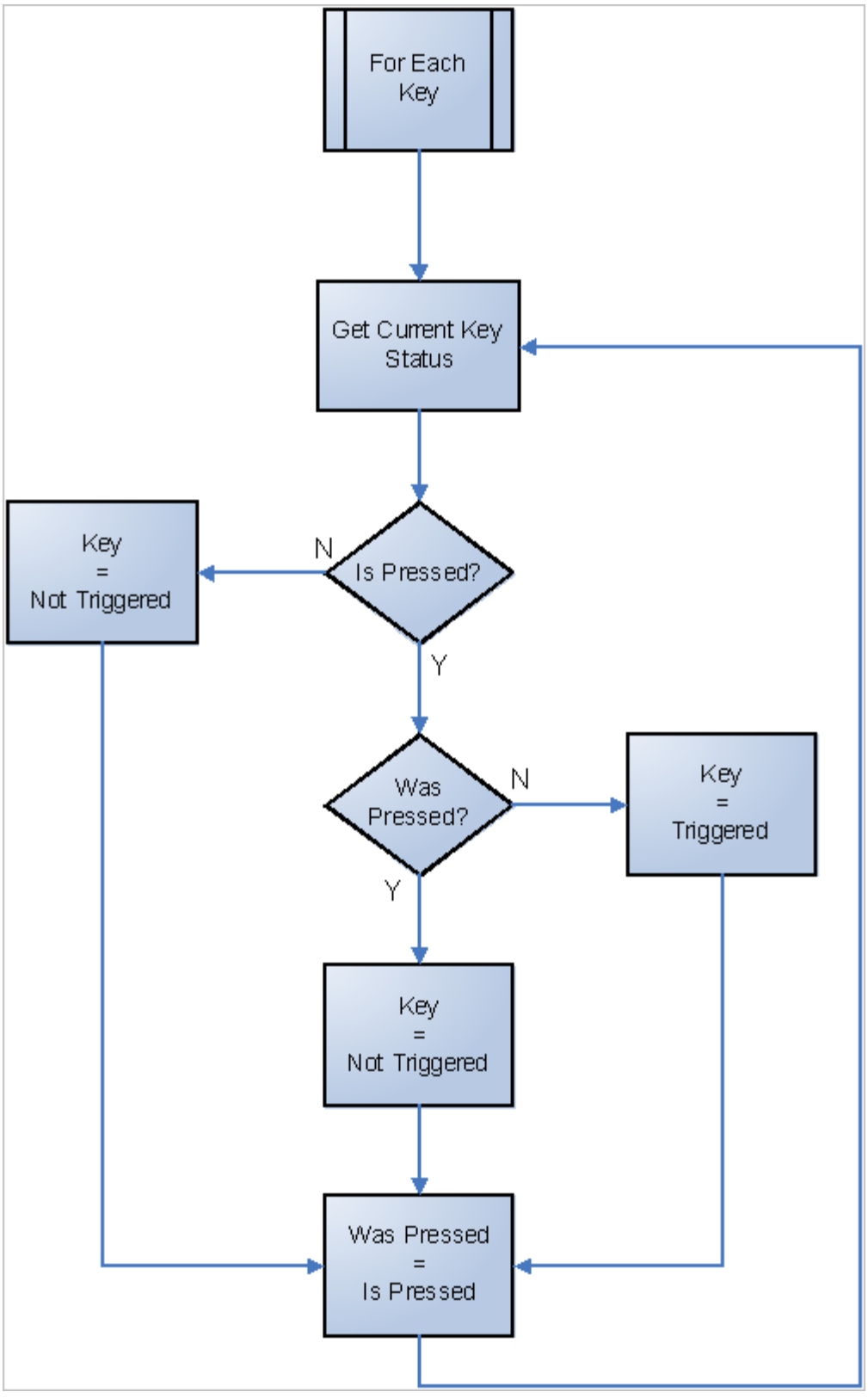
- In most 2D engines, the main purpose of a graphics manager is to load and unload texture files.
  - It can also be used to load 3D meshes, which are files containing everything that is required to render back a model properly in the game, like a list of vertices, their colors, texture coordinates, face indices and other components. (The vertex structure will be explained later on).
- Although game objects use textures, they should not be responsible for loading and unloading these textures.
  - Remember that game logic should be totally separated from the hardware, and that system components should be the only "communication channel" between game logic components and system components.
  - Textures should be loaded in the graphics manager (which is a system component), and game object instances should point to the textures they need.
  - This has another major benefit, which is preventing a texture from being loaded more than once, which drastically decreases the amount of memory used by the application.
- In a DirectX application for example, the graphics manager would be responsible for initializing a Direct3D object, a Direct3D device, and set up functions to load, render and unload textures and meshes.

- **Audio manager**
  - Most games play some kind of music that keeps looping throughout the game.
  - Some sound effects are played at special game situations (Example: "Win music" is played when the player completes a level...).
  - Usually, we don't have lots of sound related functionalities besides Play, Pause and Stop. Therefore it's good to have an audio manager which simplifies sound playing tasks.
  - An audio manager should have these functionalities:
    - Play background music (Could be streaming the music or playing a midi file)
    - Play multiple sound effects at the same time.
    - Be able to transition smoothly from one background music to another.
    - Adjust sound effects/music parameters based on what is happening in the game.
  - A basic audio manager should implement at least the first 2 aforementioned features.

- **Input manager**
  - Usually it's fairly simple to directly read user input from the hardware, although it is not a good practice.
  - It's better to address input devices within the input manager, and have other components inquire input from the input manager (And not directly from the input device)
  - Encapsulating the input device into the input manager has other benefits like:
    - Standardize input data. All components will be using the same data format.
    - Reading input is done once per frame. All components will be using same input data.

- Perform a data post processing to simplify any future data requests. Post processes allows the input manager to:
  - Track currently pressed keys
  - Track the newly pressed keys, or in other words the “Triggered” keys. These keys are pressed now but they weren't pressed during the previous frame.
  - Track the newly released keys, which are keys that were pressed during the previous frame but were released this frame.
  - A history of pressed keys
- Minimize the changes in other components when a new input device is used.
  - For example, although reading input from a joystick can be very different from reading input from a keyboard, game components should inquire about both devices' input using the same convention.

- Usually, the hardware allows us to determine the current status of keys and buttons.
- Unfortunately, this information is not enough for games and applications in general.
- What if you want to shoot just 1 bullet when a certain key is pressed? Although that key will be pressed for a short amount of time, several bullets will be shot because several game loops were completed during that short amount of time.
- This problem can be solved by implementing a trigger functionality for keyboard keys, mouse and joystick buttons etc.. Now each key/button has 2 different states: Pressed and Triggered. If a key is pressed from loop n until loop n + m without being released, the “Pressed” state of the key will be enabled during all these loops, but the “Triggered” state will only be enabled during the first loop, which is n.
- In order to implement the trigger function, we have to save the status of each key during the current loop in order to compare it to the same key's state during the following game loop.
  - If a key is newly pressed (which means it wasn't pressed during the previous frame), it will be considered as both “Pressed” and “Triggered”.
  - On the other hand, if it is pressed during the current and previous frame, then it will be considered as “Pressed” only.



1.5. Game Logic Components

- **Object manager**
  - All games have multiple objects located throughout the game world, each being updated according to its own defined behavior.
  - These objects are saved in a single list called the "object list" which contains all the object instances of the current game state.
  - It is not unusual for an object list to contain hundreds, if not thousands of objects.
  - These objects interact with the player and with each other. Ultimately this interaction defines the gameplay. Examples:
    - The "cars" in a racing game.
    - The "weapons & items" in a role playing game.
    - The "AI controlled enemies" in a first person shooter.

  - There are few things that could be considered as objects, but not managed in the object manager for one reason or another.
  - For example, the environment or any static object is not managed by the object manager because they practically do nothing but being drawn. (More on that in the environment manager). Therefore the object manager handles only the objects that are "alive" in the world.

  - The object manager is responsible for:
    - During game state load:
      - Loading all objects data (Textures, meshes, AI behavior...) of the current state and initialize them properly.
    - During game state initialize:
      - Loading the list of objects instances used in this state (The instance of each object).
      - Filling the object list with the object instances.
    - During game state update:
      - Updating all objects instances found in the object list.
      - Creating new object instances and adding them to the object list depending on the game logic and input.
      - Removing "dead" instances from the list.
    - During game state draw:
      - Send all the objects to the graphics manager in order to be drawn.
    - During game state free:
      - Remove all remaining objects instances and freeing their allocated memory.
    - During game state unload:
      - Unload all object material that was loaded in the game state load.

- In order for the object manager to manage the objects, the object instance entity needs to provide some functionalities:
  - Clone: Used to create an exact copy of the object instance and return it. This is mainly used in games where lots of instances share the same object type. Example: “Bullets” in a shooter game.
  - Initialize: Used to reset the object instance to its initial state (Reset the animation counter to 0, reset the instance position...)
  - Update: Used to update the object instance according to its own behavior.
  - Render: Used to render the object instance.
  - Free: Used to free the object instance's resources prior to being deleted.

- Notice that there are no load/unload functionalities for the object instance entity, since it is not responsible for loading and unloading its rendering/AI... data.

- **Environment manager**
  - As explained previously, the object manager takes care of object instances that are “alive” in the world, which in other words are the instances which the player can interact with.
  - A game usually contains object instances of a different kind, which are static objects.
  - These object instances can't be interacted with; their only purpose is to make the game scene look pretty and more appealing for the player. Examples:
    - Sun/Moon in an open area game.
    - Birds in a real time strategy game.
    - Grass in a racing or shooter game

  - These type of object instances are handled by the environment manager. The main task of the environment manager is to determine which static instances are visible and send them to the graphics manager component in order to be drawn.
  - Since these objects instances are static, they are usually preprocessed during the initialization of the game state in order to reduce the amount of time spent at run time.
  - Note that when we said that the player can't interact directly with these static objects, that doesn't mean that they can't be affected by the player.
    - For example, if a game has a moon and a sun rotating in the sky, their positions are dictated by the game time.
    - While playing the game, the player could obtain special powers that allow him to speed up or speed down the game time, thus indirectly affecting the position of the moon and the sun.

- The environment manager should have these functionalities:
  - Load: Load environment data (rendering material...). This is called once while loading the current game state.
  - Initialize: Initialize the instances to their initial states (Initialize the animation counter to 0, initialize the static instances positions...).
  - Update: Update the environment data (Some instances positions should be updated, like the previous moon/sun example).
  - Draw: Send the visible environment data to the graphics engine.
  - Free: Remove the static instances and free their allocated memory.
  - Unload: Unload all the static objects material that was loaded during game state load.

- **Game state manager**
  - A game is always in a state. A game state could be “In Main Menu”, “In Level 1”, “In Loading Screen”..
  - The game state manager is responsible for game state switching.
  - Handles allocation/deallocation of each games state's memory pool.
  - Handles the game loop, which contains the Update/Draw functions of the current state.
  - Handles the frame rate controller.
  - Each state is associated with a set of functions that manages that state's cycle
  - These cycle functions are:
    - Load: Loads the state's necessary data and initializes it. This function is called only once at the start of the state. If the state is restarted, its data should not be dumped and loaded again, therefore the “Load” function should not be called upon restarting a state.
    - Initialize: Used to prepare the state's data in order to be used for the first time. It loads no data whatsoever. If a state is restarted or reset, this cycle function is used.
    - Update: Updates the state's data based on several factors like user input, time or gameplay logic...
    - Draw: This cycle function sends the state's “draw data” to the graphics engine component.
    - Free: Used to clean up the state and make it ready to be unloaded or initialized again. No data is dumped in this cycle function.
    - Unload: This cycle function is called when the state should be terminated (Not restarted). It dumps back all the data that was loaded in the state's load cycle function.