CPSC 4160/6160 Fall 2022

PROJECT 2

Due: In-class demonstration on October 25, 2022; code delivered to handin by end of day, October 25, 2022

DESCRIPTION:

This project is to build a very basic and limited game engine, and implement a simple game called “WhackABox”. The purpose of the game is to verify and demonstrate the performance and execution of the game engine, with the new elements added in this project. This project is worth 15 points, divided up broadly into three categories:

1. Does it work and are the specific requirements of the assignment implemented? (5 points)
2. Was the demonstration informative about the strengths and weaknesses of your project? (5 points)
3. Is the code clean, organized, and demonstrative of pride of craftsmanship, and easy to execute? (5 points)

On October 25, you will submit this project for grading in two ways:

1. In-class demonstration: You will have 1-2 minutes with either the instructor or TA to demonstrate, live, your game playing. You should show all of the features of gameplay asked for in the description below. This is your chance to present your project in the best light possible.
2. By the end of the day, October 25, you must submit the code for your project to the SoC handin system. The specifics for doing this are posted on the website for the class. Pay careful attention to this task: many students who were very competent have messed up this task and suffered point loss. We will expect to be able to run the game on our own machines, as well as inspect the code.

GAME ENGINE

As we have discussed in class, our python-based game engine must have the following organization of directories (`gamey` is the name of my game engine code - feel free to give yours any name that you like, while retaining this directory organization inside):

```
gamey
├── assets
│   ├── sounds
│   └── textures
├── engine
│   ├── actor
│   │   ├── action
│   │   └── entity
│   ├── asset
│   │   ├── action
│   │   └── entity
│   ├── crowd
│   │   ├── action
│   │   └── entity
│   ├── enviro
│   │   ├── action
│   │   └── entity
│   ├── fx
│   │   ├── action
│   │   └── entity
│   ├── physics
│   │   ├── action
│   │   └── entity
│   ├── play
│   │   ├── action
│   │   └── entity
│   ├── render
│   │   ├── action
│   │   └── entity
│   ├── sound
│   │   ├── action
│   │   └── entity
│   ├── story
│   │   ├── action
│   │   └── entity
│   ├── ui
│   │   ├── action
│   │   └── entity
│   └── utility
│       ├── action
│       └── entity
├── python
└── templates
```

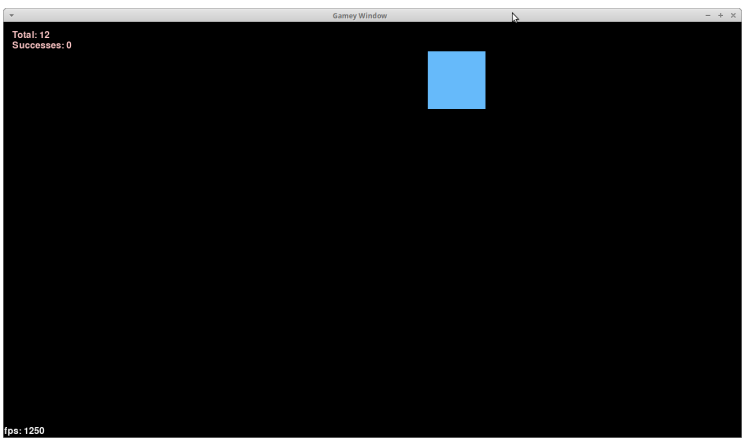
Do not forget that inside each directory, there needs to be a valid python file called "__init__.py". An empty file is valid, but you can also use it to create utility or factor functions, for example.

If you find that you want to implement more actions and/or entities than described here, feel free to do so. However, you should be careful whether or not more are actually needed for the game that must be implemented. More can certainly be useful in the long run for the game engine, but pace yourself and make sure you get the assigned project done.

Final word of advice, paraphrasing Gandalf: "Keep it simple, keep it safe".

WHACKABOX GAME

The Whackabox game is motivated by the classic whack-a-mole arcade game. A basic rectangular button appears at some random location in the game window, and remains at that location for up to a limited amount of time. If the player clicks the button before the time elapses, the player gets a point. After the successful click, or after the time limit expires, the button disappears and reappears at a different random location. A counter keeps track of successful button clicks, and a second counter keeps track of the number of times the button has appeared. Both numbers are displayed in the upper left corner. Here is a screen shot of the instructor's implementation:



On startup of the game, the window should be sized 1280 x 720, or whatever size best fits your display, and remain that size throughout the game.

At any time, pressing the escape key or the quit button on the window should end the game and python should terminate.

The game code itself should be contained in the "python" directory, called "whackabox.py", as a python script that imports and uses the various elements of the game engine to initialize the game, entities, actions, and then runs the game loop.

This game requires the following entities in the following engines:

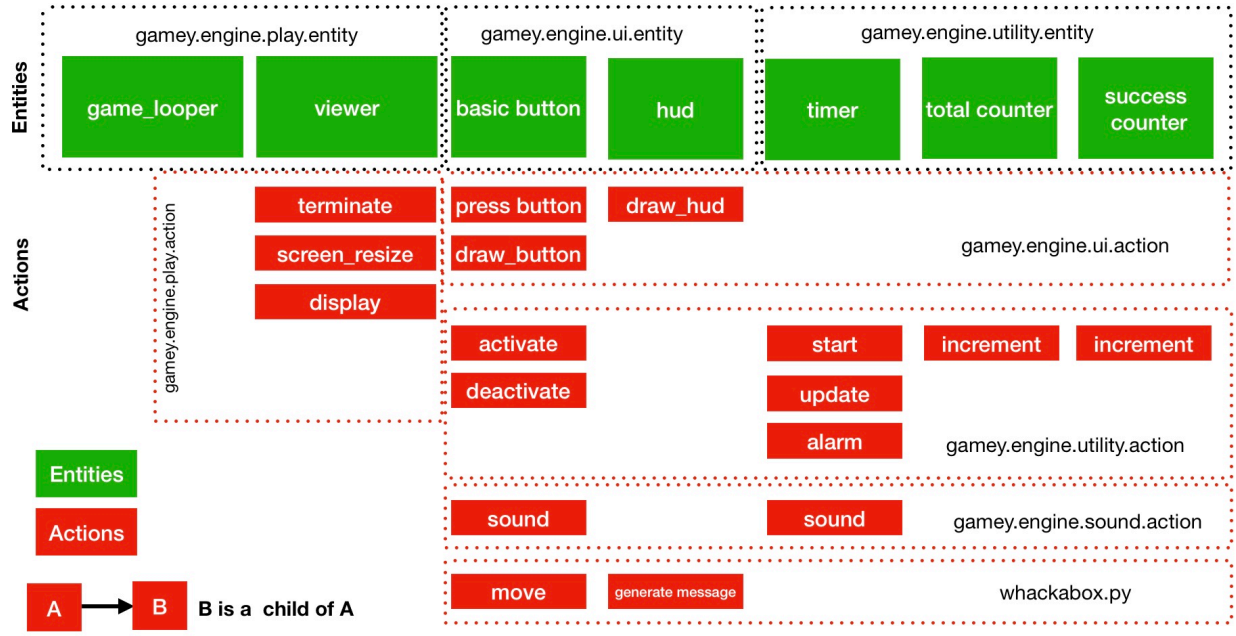
play - game_looper and frame_viewer, just as in the first project
ui - button, hud (heads up display)
utility - timer (keeps track of time passage), and counter (increments an integer counter)

and the following actions in the following engines

play - terminate and screen_display, as in project 1
ui - press_button, draw_button, draw_hud
sound - emit_sound
utility - start_timer, update_timer, alarm (with input of how much time elapsed before the alarm is triggered),
counter_increment,
activate_entity,
deactivate_entity

In addition, there are two actions that are custom to this game and not suitable for inclusion in the game engine. They can be contained in the script whackabox.py. The entire collection that must be implemented for this game is shown in this diagram:

Whackabox



Actions listed under an entity use that entity as their entity_state. The following is a description of each of these entities of the game:

1. The game_looper and frame_viewer entities and actions are the same kind as with the first project. The associated actions are the same as previous project(s).
2. There is one button and one hud (heads up display). The button is very basic: its draw just displays a rectangle with color.
3. The hud displays two lines of text in the upper left corner of the window. The first line shows the value of the counter that counts how many times the button has appeared. The second line shows the value of the counter that counts how many times the button has been successfully clicked.
4. The timer tracks the current time, in milliseconds, since the start of the game. On construction or start, the time is stored in a class variable as the “start_time”. A class method “tick” updates the class variable storing the current time. A class method “elapsed_time” takes the difference between the current time variable and the start time variable and returns it.
5. The total_counter counts the total number of times that the button has appeared in the game window.
6. The success_counter counts the number of times that the player successfully clicks the button before it disappears.
7. The action press_button handles the MOUSEBUTTONDOWN pygame event and determines whether or not the button was pressed. Code for this has been presented in class and in the UIstuff.pdf document. This action is of type “event”.
8. The action draw_button draws the rectangular button in the window at its current location. Code for this has been presented in class and in the UIstuff.pdf document. This action is of type “display”.
9. The action draw_hud draws its designated lines of message on the window. This action is of type “display”.
10. The action “activate” sets the value of the the active variable in its entity_state to True.
11. The action “deactivate” sets the value of the active variable in its entity_state to False.
12. The action “sound” plays a sound. This action occurs in two different spots in the diagram, and each of these should emit a different sound.
13. The action “start” starts the timer, resetting its start_time variable to that moment.
14. The action “update” updates the current_time variable of the timer to the current time. This action is of type “loop”, so it acts in every pass of the game loop.
15. The action “alarm” tests whether the timers “elapsed_time” has exceeded the allotted time stored in the alarm, and executes child actions when exceeded. The allotted time is an input to the constructor and is stored as a class member.
16. The action “increment” adds 1 to the value of the counter in its entity_state.
17. The action “move” is custom to this game and so is not in one of the engines. When it acts, it changes the location of the button to some random location in the window, and it changes the color of the button to some random color.
18. The action “generate_message” is custom to this game, and so is not in one of the engines. It uses the values in the total_counter and success_counter entities to update the two message lines of the hud to the current values of the counters.

All actions, once they have acted with their specific purpose, also call all of their children to act. Except for the few actions that were explicitly assigned a type in the list above, all actions are NOT “display”, “event”, or “loop”.

This game works by suitably connecting the various actions to each other, so that some actions children of others. Note that in the UIstuff.pdf file a similar diagram was shown for a different game, with a set of buttons and a hud, and arrows drawn between actions to show which ones where children of others.

This current project can be accomplished by doing two fundamental tasks:

1. Figure out the pattern, in the diagram above, of which actions are children of which other actions.
2. Implement the various actions and connect them according to task 1.

Keep in mind that each of these actions is a very small amount of code.

STUDENTS IN 6160:

If you are in 6160, there are additional implementation elements. If you are in 4160, you may implement these also because you may find them useful. But 4160 grades will not be impacted by this additional detail.

The allotted time in the alarm is an input, nominally at constructor time. It's value can also be changed at any moment because is it stored as a member of the alarm action. Rig your game so that each time the alarm starts again, the allotted time is reduced slightly, so that as the play progresses, the time that the button is available to press slowly diminishes, increasing the difficulty of the game. However, you cannot do that by added this logic to the alarm action. It must be done by adding a one or more new action(s) that decrement(s) the allotted time just before the timer start action takes place.