Tutorial – Sequence Diagrams

Introduction:

In this tutorial (and in the tutorials for the remaining UML sessions) we are going to analyse a simple game so that we can document its design using UML diagrams.

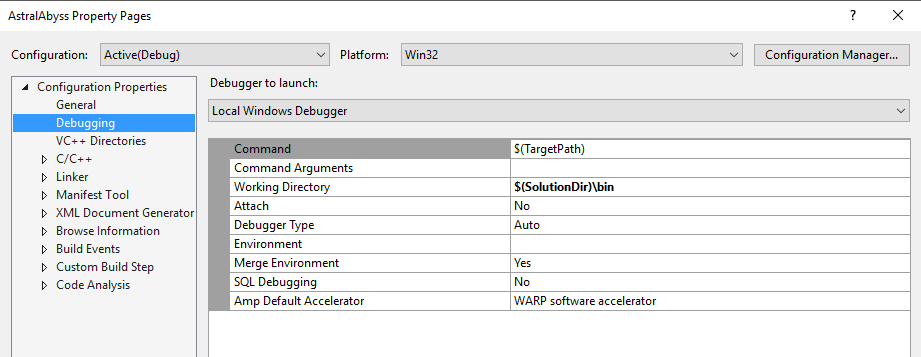
In this tutorial we will create a sequence diagram for the game Astral Abyss.

In the tutorials for previous UML sessions you would have created various diagrams describing this game. Completing these exercises and tutorials will assist you in this tutorial, so you may find it useful to complete them first if you have not yet done so.

Set Up:

The Astral Abyss project is available on the Resources page for this subject. If you have already added this project to your *aieBootstrap* solution, you can skip this step.

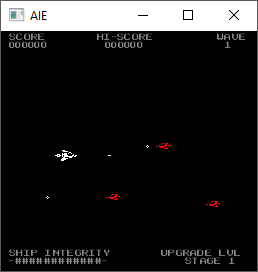
The game is provided as a project that will link into the *aieBootstrap* solution. If you do not yet have a copy of *aieBootstrap*, you will need to download that from this github repository: <https://github.com/AcademyOfInteractiveEntertainment/aieBootstrap>

1. Download *AstralAbyss.zip* from the *Resources* page for this subject
2. Extract the zip file to your computer. (A good place to extract it would be to the bootstrap solution folder)
3. Open the *aieBootstrap* solution
4. In Visual Studio, add the *AstralAbyss* project to the solution.  
   In the *Solution Explorer*, right-click on the solution and select *Add -> Existing Project*
5. Open the properties for the *AstralAbyss* project and ensure the debug *Working Directory* is set to **$(SolutionDir)bin\**  
   
6. Lastly, we need to copy the images and fonts this project uses to the solution’s *bin* folder.

In the AstralAbyss project folder you will find a *bin* folder containing images and fonts. Move these into the **$(SolutionDir)\bin** folder.   
  
If your game launches and you cannot see anything drawn, you have likely copied the resources to the wrong folder.

Once you have set up the project, set it as the active *Start Up Project*, compile and then run the project.

You should be able to launch and play the *Astral Abyss* game without errors.



Creating a Sequence Diagram:

We are going to create a diagram detailing the sequence of messages (function calls) that occur when the player presses the space bar to fire.

Before starting, you should review the source code of the game so that you are familiar with how the shooting mechanism works.

We’ll start by listing all the objects that are involved with this event.

For this, I’m going to largely ignore the player and how the *Input* class actually receives the input. Obviously the ‘press the space bar’ even is initiated by the player, but we aren’t dealing directly with the callback functions that are executed in response to this event.

Instead, the game is calling the *Input* class’s *isKeyDown()* function every frame to check if the desired key is being pressed.

If the game was processing input by implementing callback functions, we might model this differently.

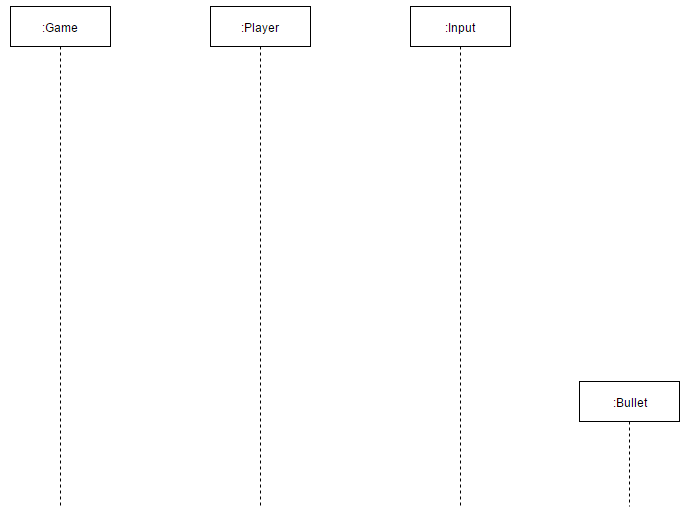
To model the ‘press space to shoot’ event, we’re instead going to start with the *Game* class’s *updateGamePlay()* function, since this is where the game checks to see if the *Player* class needs to spawn a bullet and spawns one if required.

The list of objects (classes) that we will map in this sequence diagram are:

* The Game class,
* The Player class,
* The Bullet class, and
* The Input class.

Remember, this is an abstraction of how bullets are fired only. So even though we will be drawing the *updateGamePlay()* function in our diagram, we’re only going to concern ourselves with the parts of this function that relate directly to firing a bullet.

With our list of classes defined, we can start creating our sequence diagram.

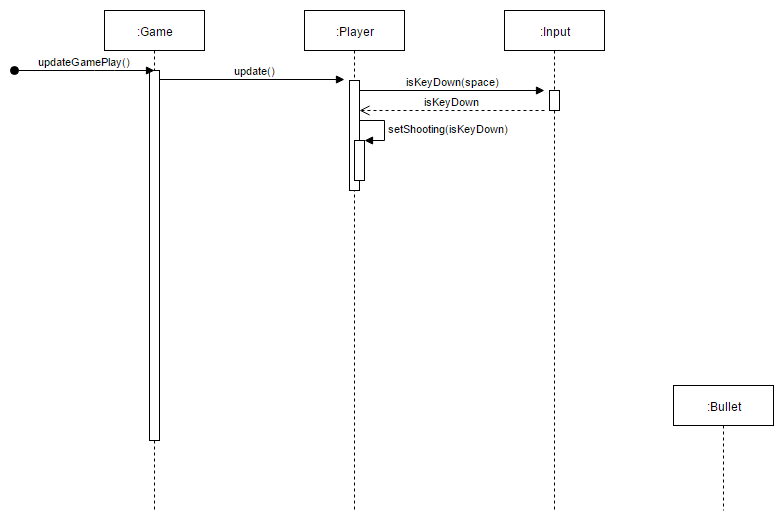


There are two things to note here. First, the lifelines for all objects extend to the bottom of the diagram. Second, only the Bullet class box is not aligned at the top with the others. This indicates the instance of the Bullet class shown isn’t created until a later point in time.

If you look at the *Player’s update()* function, you’ll see that it calls the *Input* class’s *isKeyDown()* function to see if the space bar was pressed. If it was, it sets its *m\_isShooting* variable to true.

The *Player’s update()* function is called by the *Game’s updateGamePlay()* function – which is the function that will be called initially.

Let’s model all that information in our diagram now:



Take a moment to relate what’s happening in our updated diagram with what you can see in the code.

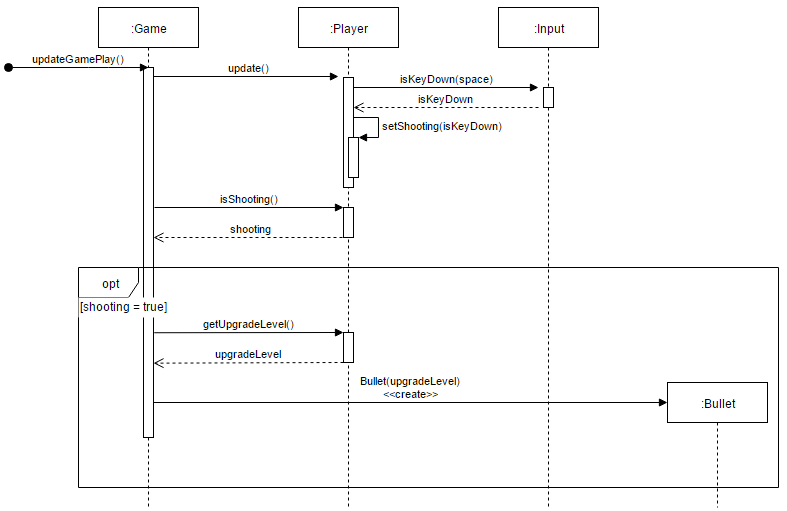
We don’t care what is calling the *Game’s updateGamePlay()*, only that it is the first function called for this event.

Also, in the code you will see that the *Player’s* *update()* function simply sets the value of *m\_isShooting* directly. I’ve shown this as a self-call to a function called *setShooting()* to highlight that a state change has occurred. Remember, this is an abstraction and its acceptable to remove or add detail if it helps clarify your design.

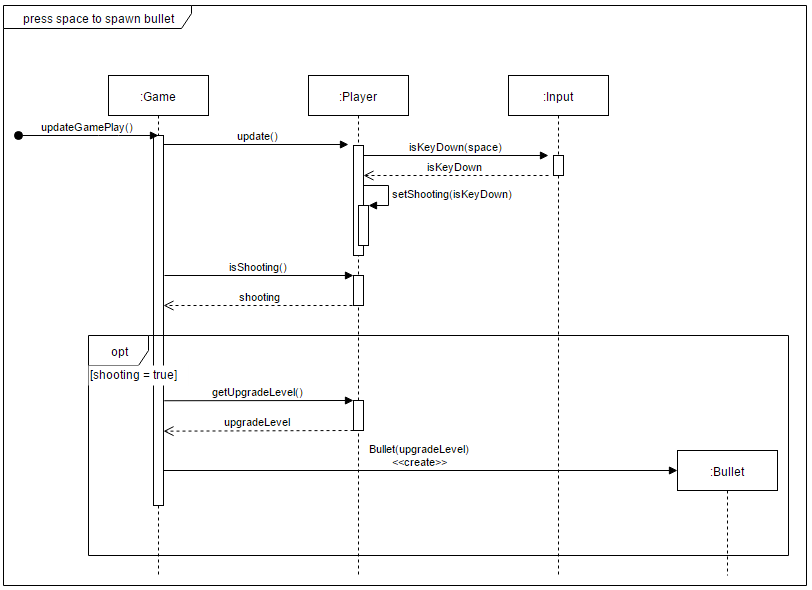
In the remaining part of our diagram, we want the *Game* object to check if the *player* is shooting and, if so, spawn a new bullet.

We’ll make another call from the *Game* object to the *Player* object to see if it is shooting. Then, we’ll use an option frame (opt) to highlight that the bullet is created only if the call to *isShooting()* indicated the *Player* is shooting.

The remainder of the diagram is drawn as follows:



Finally, it’s a small detail and easily forgotten, but you will want to label your diagram to explain what it is showing. You can do that using a frame:



Exercise:

Find another event in the game and map that using a sequence diagram.

If you can’t find one yourself, then create a sequence diagram that shows what happens when a bullet collides with an enemy ship.