Tutorial – Activity 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.

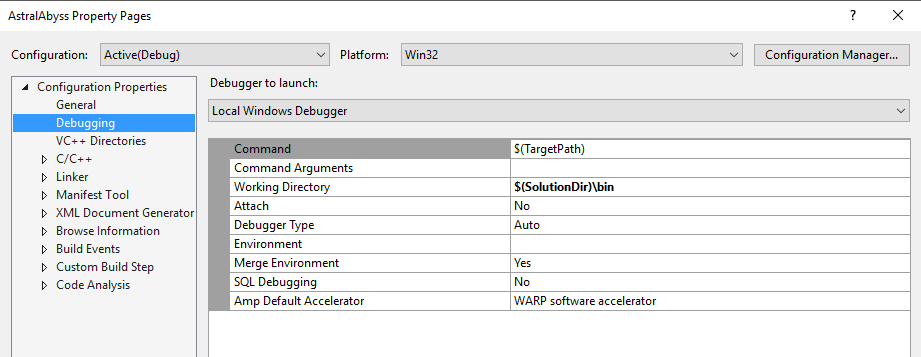
In this tutorial we will create an activity diagram for the game Astral Abyss.

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.

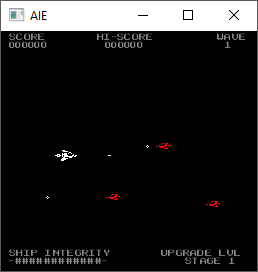
The game is provided as a project that will link into the *aieBootstrap* solution. If you do not yet have a copy of *aieBootstrap*, you will need to download that from this github repository: <https://github.com/AcademyOfInteractiveEntertainment/aieBootstrap>

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 an Activity Diagram:

Activity diagrams describe the flow of control of your program. They are very similar to flow charts (which many people already have experience with), except they allow for the inclusion of concurrent processes in your design.

The *Astral Abyss* program contains no concurrent processes, so for our purposes the activity diagram is practically the same as a flow chart.

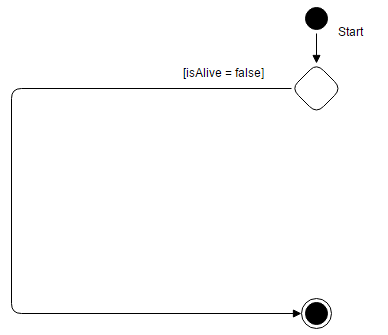
These diagrams can be used to depict any level of abstraction. We could create a very high-level diagram to describe the whole game at a very broad level of detail, or we can be very specific and describe the operation of a single function.

Generally, we would aim for somewhere between these two levels of detail. Although describing individual functions using these diagrams may be useful if we are doing something particularly complex (say, the AI algorithm of an enemy).

For this tutorial we are going to draw a diagram explaining the *update* function of the *Player* class.

If we look at the *Player* class’s *update()* function, you can see that the very first thing the function does is check whether or not the player is alive. If the player is dead, the function exits straight away.

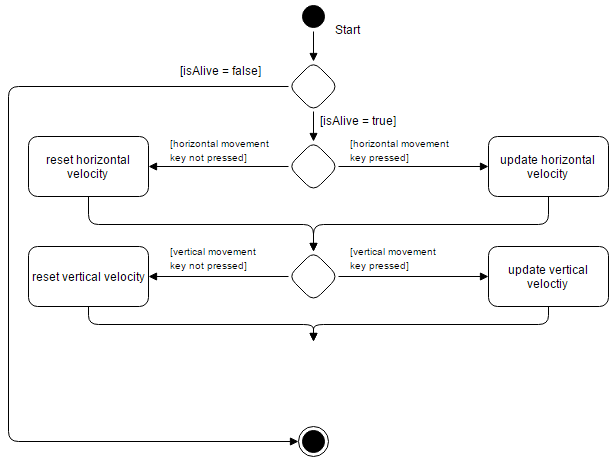
We can start by drawing this logic in our *activity diagram*.



The next processing that happens in this function are some checks to see if any direction buttons were pushed. If a direction key was pressed, the velocity of the player is updated by the appropriate amount.

Because putting in 4 conditions (one for each direction) is a bit tedious, I’m going to merge this processing into two checks. So in my diagram I will have checks for ‘was a horizontal movement key pressed’ and ‘was a vertical movement key pressed’, rather than checking each direction individually. We could have also combined all 4 checks into a single ‘was any movement key pressed’ check instead. Remember that these diagrams are abstractions and we can add as much (or as little) detail as needed, as long as our design intent is clear.

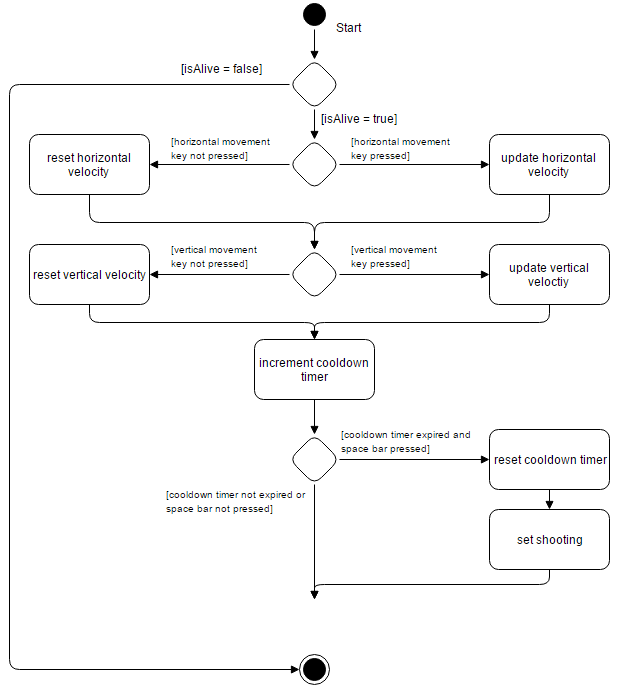
Here is the updated diagram with the direction key checks:



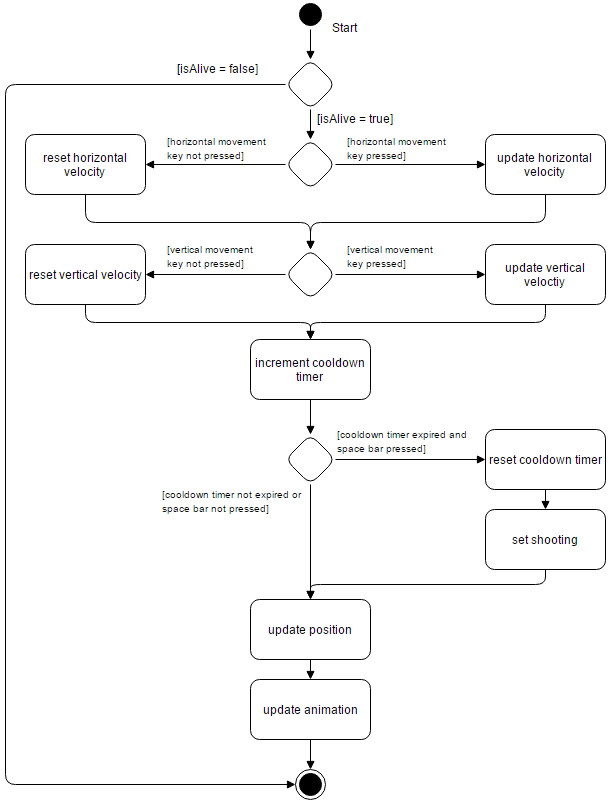
Following the direction key checks we have a check to see if the player is shooting.

Shooting happens when the player presses the space bar, and when the ‘cooldown’ timer has reached a certain value. The cooldown timer prevents the player from spamming the fire button and creating an endless stream of bullets.

If the checks for firing all pass, then we reset the cooldown timer and set the player’s *isShooting* variable to true. In our diagram we can combine the two conditions (has cooldown timer expired and is space bar being pressed) into the one condition:



The last thing to do is to update the position of the player according to the new velocity, and update the animation. This code is actually contained within the *Entity* base class. I’ve chosen to include the complete logic in this diagram (since its small), but we could have alternatively had a box that said something like ‘perform base class update’. (If you wanted to be really detailed, you could then include an *activity diagram* depicting this base class logic).

Here is the final diagram:

Exercise:

Typically, you would only create an activity diagram for parts of your program that may be difficult to understand by reading the code alone, or when you want to provide more clarification on the intended design.

These diagrams can also be abstracted to any level (for example, very general, very specific, or anything in between). You can also have nested diagrams (for example, we could break our ‘update animation’ task in the diagram above into its own diagram)

Perhaps the most complex logic in this game is contained in the *Game* class’s *updateGamePlay()* function.

Either individually or in groups, create an activity diagram for the *updateGamePlay()* function.