Opleiding Informatica

Gradual OverPy: a proof of concept of applying a gradual method on the Overwatch Workshop

David Lin

Supervisors:
Felienne Hermans & Giulio Barbero

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

20/07/2021

Abstract

This is where you write an abstract that concisely summarizes your thesis. Keep it short. No references here — exceptions do occur.

Contents

1 Introduction

In this section an introduction to the problem addressed in this thesis is given.

1.1 The situation

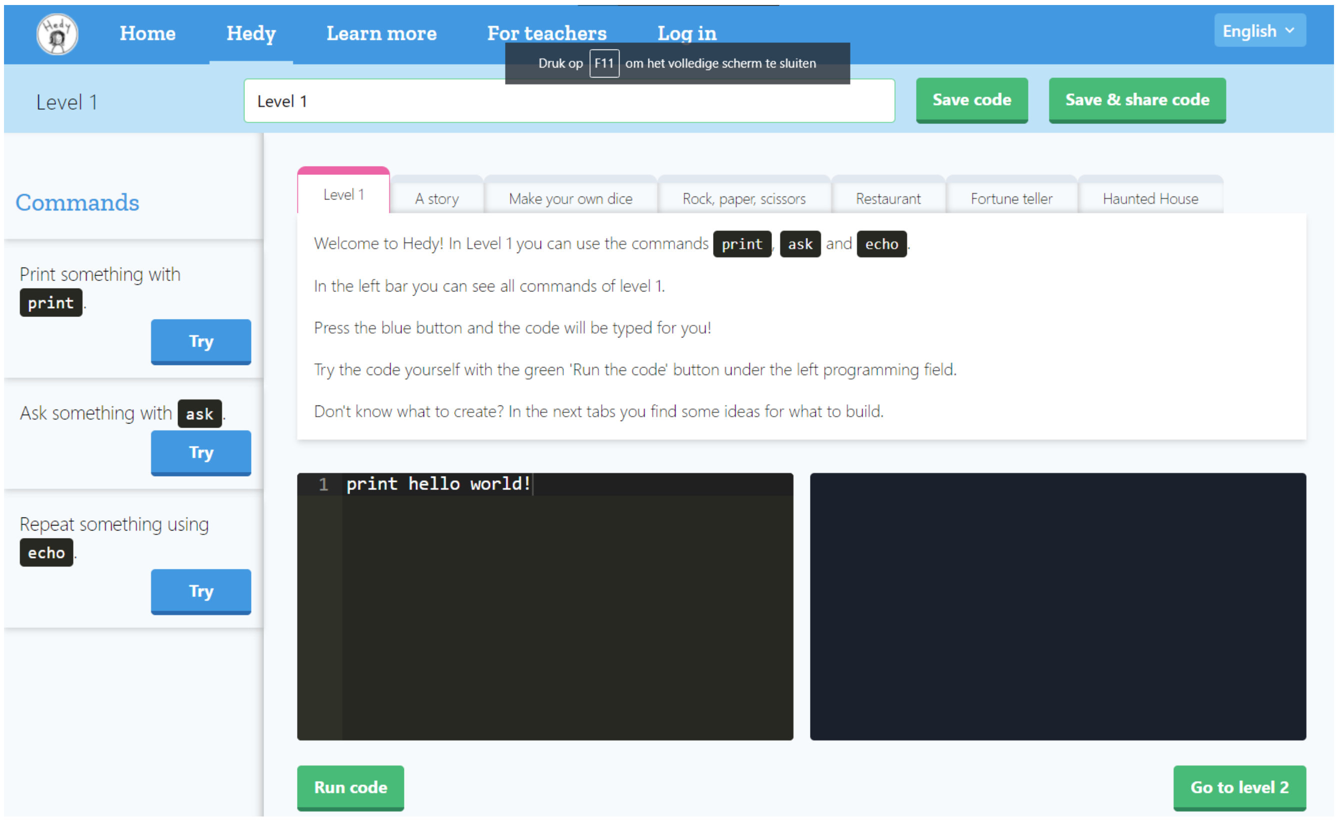
In the current era of technology, there is a need for programmers in the world [Her20]. You could argue that there is no end to the demand for client-customized software, as people have very specific needs. One of the problems with current programming learning methods is that complete online tutorials like Geeksforgeeks[1] can be overwhelming for a beginner as there are many concepts to understand, which makes learning a programming language less appealing.
In the case of either learning a programming language through a simplified online tutorial or through a course, an aspect that people struggle on is the syntax of a language. [Her20] Learning programming syntax is said to require "a level of attention to detail that does not come naturally to human beings" [CM08]. This means that students are not guaranteed to be able to learn such syntax, the potential consequence being an increasing rate of these students dropping out from their Computer Science studies increases.
An approach to provide a proper chance to these students has been made by Hermans et al. with their web application called Hedy[2] (see Figure 1) [Her20]. They make programming in Python more accessible with their gradual language learning method. This method puts the focus of programming language learning on different aspects at the time [Her20], starting with the introduction of functions with simple syntax and later on introducing a gradually changed syntax.
However, a disadvantage of learning programming with such language is that the visible result of your code can only be seen in text form (see Figure 1). To make this process more interactive, this paper introduces gradual learning on the mod workshop of a multiplayer first person shooter called Overwatch (PC version) [Ent16], in order to allow players to play and interact with their code.
Even though Overwatch has an integrated overlay (see Section 2.3) and this overlay is manageable for less experienced programmers, it does come with the downside that there is no actual programming involved, as the overlay allows the user to add, remove and edit elements by simple clickable interaction.
However, the game also allows to import code through copying and pasting (which is the reason the PC version of the game is used). In this paper, a web application has been developed that users can utilize to learn how to program in this workshop. It involves 3 main elements:

- OverPy [OvPa], a script that simplifies the syntax of the original workshop code
- The gradual introduction of elements with explanations and examples
- Online integrated editor,which the users can use to create their workshops

The research question is as follows:

Does a gradual language system help Overwatch players develop custom workshops?

[1]Geeksforgeeks can be found on: https://www.geeksforgeeks.org
[2]Hedy can be found on: https://www.hedycode.com



Print something with `print`.

Ask something with `ask`.

Repeat something using `echo`.

Welcome to Hedy! In Level 1 you can use the commands `print`, `ask` and `echo`.

In the left bar you can see all commands of level 1.

Press the blue button and the code will be typed for you!

Try the code yourself with the green 'Run the code' button under the left programming field.

Don't know what to create? In the next tabs you find some ideas for what to build.

```
print hello world!
```

Figure 1: Hedy, a gradual programming language [Her20]

1.2 Thesis overview

The further sections will go into more detail what was discussed in the introduction. Section 2 provides information about what a gradual language is, why the Overwatch workshop was chosen instead of any other workshop and what (Gradual) OverPy is. In Section 3 the specifics on the developed web application are provided, including the framework and customized grammar. Section 4 shows the results of the interviews with volunteers (varying from inexperienced to experienced programmers) who tested the web application and provided their opinion on the concept of such method, after which the conclusion is drawn and future work is discussed in section 6.

2 Background

Learning a programming language is quite a task. The concept of learning such a language is similar to learning a natural language [Her20]. Elements like order of words in a sentence can be compared to syntax, a person's vocabulary is similar to a person's knowledge of existing functions, etc. When looking at language education for young children, the process of learning rules and words in a natural language follows a specific structure [Her20]: It starts with the introduction of words in lowercase letters, after which sentences are made with lowercase letter words. Only after mastering this concept by lots of repetition are uppercase words and punctuation introduced. The importance is that concepts that are introduced early should be the baseline for later concepts.

2.1 Gradual language

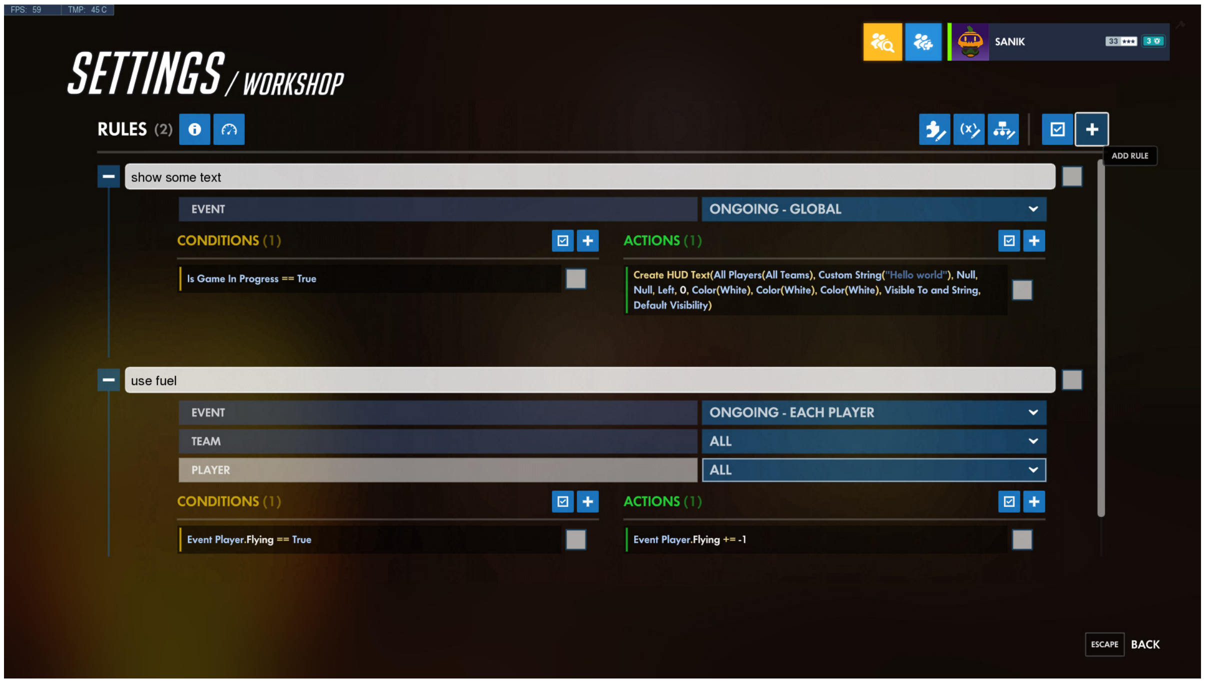
A gradual language, which was developed by Hermans et al., is a new way of learning a programming language. Instead of introducing new functions of a programming language whilst assuming the user has the insight to understand the syntax, this method simplifies the syntax to put the focus more towards functions at the first levels [Her20]. This process can be compared to simplified grammar in a natural language. After this, the syntax gradually changes to be more similar to Python, but this is done using previously learnt functions [Her20]. By introducing one concept at the time, the short term memory load of the user is not being overloaded with information at once. By reusing previously learnt functions whilst gradually changing syntax, the user experiences repetition, which is required to master different practices [Her20].

2.2 Game modifications

There is a distinct difference between programming a game as a whole and modifying one. As the development of a game is entirely from scratch and as the time and costs can be enormous, this is out of reach for most players [ENS06]. Game modifications on the other hand require no game development, as the infrastructure is already present and modifications can be accessible to many game players with the proper tools [ENS06].
El-Nasr et al. think of game modifications as a design activity, which they believe have educational benefits, the main reason being that these design activities provide engaging contexts for skill and concept exploration, which can be applied to real life problems [ENS06]. Depending on their level of interest, can it enable wide exploration within that context, after which they can utilize explored concepts to other problems in the future.
Different games require different amounts of effort to be modified. This heavily depends on how accessible such modifications are made by the developers. Some are implemented in the game itself, others require the user to install external programs. When attempting to apply gradual learning on such modification languages, user friendliness and accessibility should be considered. If a user has to install many programs before the actual learning progress, it could become less of interest.

2.3 The Overwatch Workshop

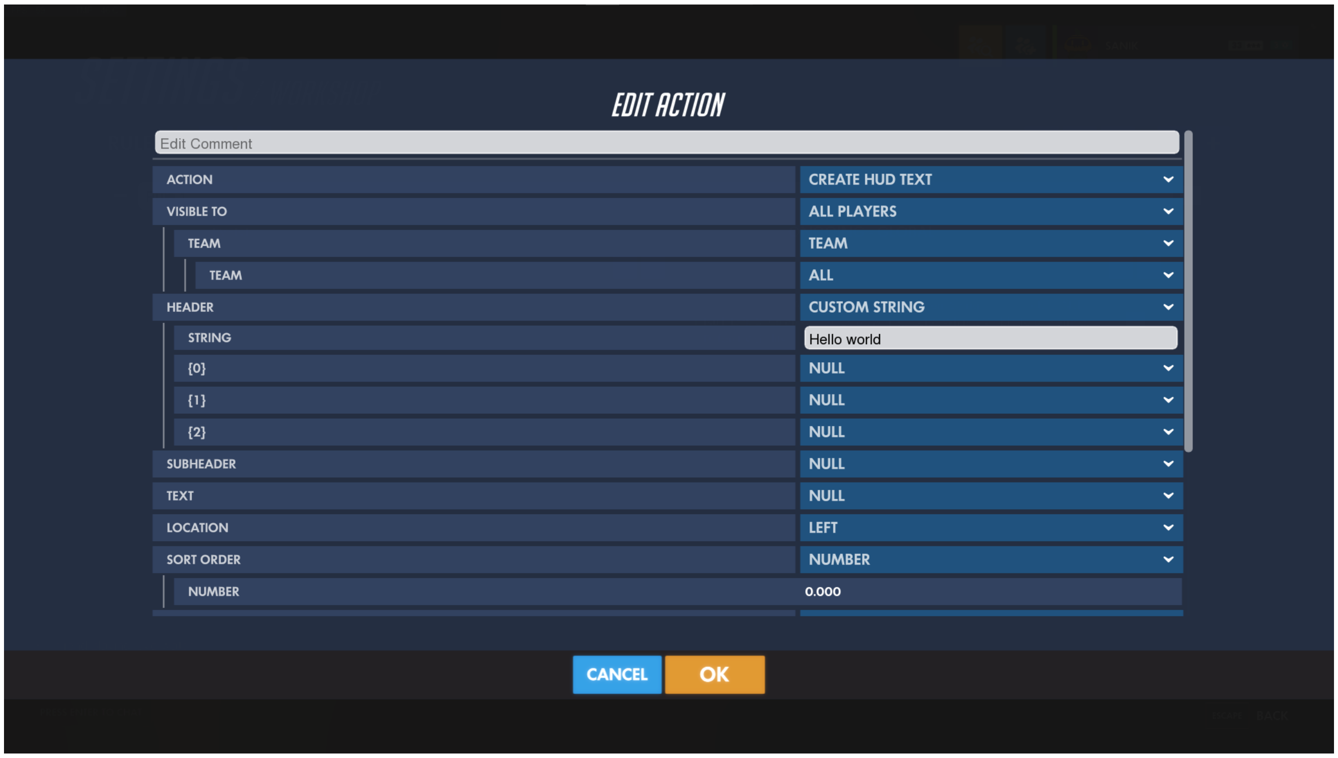
There are many different games that allow users to program their own game modes, calculators, etc., but the main reason why the Overwatch Workshop was used for this thesis is because of the ease of use, intuitiveness and simplicity.
Programming workshops, which are basically modifications to the game, requires no external programs. The Overwatch workshop is a game modification tool made by the developers, which allows players to experiment with the game and the elements it provides. The workshop has an in-game overlay made for workshop development. Creating workshops can be done without writing code by adding/modifying/removing elements with the mouse (see Figure 2 and 3).



```
show some text
  EVENT          ONGOING - GLOBAL
  CONDITIONS (1)
    Is Game In Progress == True
  ACTIONS (1)
    Create HUD Text(All Players(All Teams), Custom String("Hello world"), Null,
    Null, Left, 0, Color(White), Color(White), Color(White), Visible To and String,
    Default Visibility)

use fuel
  EVENT          ONGOING - EACH PLAYER
  TEAM           ALL
  PLAYER         ALL
  CONDITIONS (1)
    Event Player.Flying == True
  ACTIONS (1)
    Event Player.Flying += -1
```

Figure 2: Overwatch Workshop overlay

Even though creating workshops in such manner might be less time consuming than typing code, gradual learning can not/hardly be applied to such method. On the other hand, the workshop allows for workshop code to be exported to your clipboard and copied code to be imported from outside the game. This enables experienced programmers to modify the workshop code in text form. The workshop elements that are made available by the developers are quite easy to understand by their names and creating simple workshops is possible with some programming knowledge. Workshops like Counter Strike: Global Offensive [VC12] do have more possibilities, including map making, but such variety of possibilities is irrelevant when it comes to learning a programming language, as an introduction to a language does not involve that many elements.


EDIT ACTION

Edit Comment

ACTION	CREATE HUD TEXT
VISIBLE TO	ALL PLAYERS
TEAM	TEAM
TEAM	ALL
HEADER	CUSTOM STRING
STRING	Hello world
{0}	NULL
{1}	NULL
{2}	NULL
SUBHEADER	NULL
TEXT	NULL
LOCATION	LEFT
SORT ORDER	NUMBER
NUMBER	0.000

Figure 3: Overwatch Workshop overlay, create HUD text modification screen

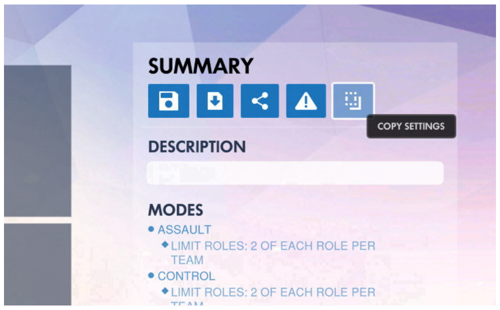

SUMMARY
COPY SETTINGS
DESCRIPTION
MODES
ASSAULT
LIMIT ROLES: 2 OF EACH ROLE PER TEAM
CONTROL
LIMIT ROLES: 2 OF EACH ROLE PER

Figure 4: Exporting Overwatch Workshop code to clipboard

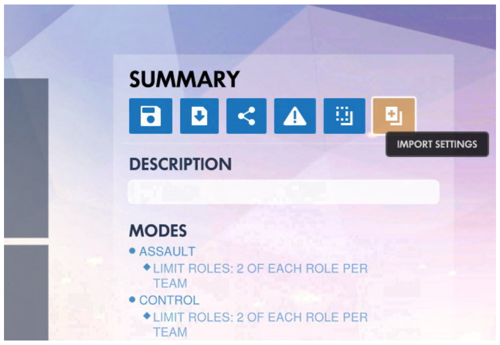

SUMMARY
IMPORT SETTINGS
DESCRIPTION
MODES
ASSAULT
LIMIT ROLES: 2 OF EACH ROLE PER TEAM
CONTROL
LIMIT ROLES: 2 OF EACH ROLE PER TEAM

Figure 5: Importing Overwatch Workshop code with code in clipboard

2.4 OverPy

The workshop has elements that are easy to understand by their names, but as for the syntax of the workshop language, it's interpretability is relatively bad compared to Python. Python uses indentation and few to no brackets, whilst the workshop code uses many brackets and functions are allowed to have spaces in their name.

```
Workshop code:
rule("show some text")
{
    event
    {
        Ongoing - Each Player;
        All;
        All;
    }
    actions
    {
        Create HUD Text(All Players(All Teams), Custom String("Hello world"),
        Null, Null, Left, 0, Color(White), Color(White), Color(White),
        Visible To and String, Default Visibility);
    }
}
```

This is where OverPy is used. OverPy is a script, made by Zezombye, that is able to transform workshop code to a more Pythonic syntax. After the code above is compiled to OverPy syntax, the code looks as follows:

```
OverPy:
rule "show some text":
    @Event eachPlayer
    @Hero all

    hudHeader(getAllPlayers(), "Hello world", HudPosition.LEFT, 0,
    Color.WHITE, HudReeval.VISIBILITY_AND_STRING, SpecVisibility.DEFAULT)
```

The OverPy code is structured less complex, rules are seperated in indented blocks instead of brackets, and the overall overview is higher level compared to the workshop code. As the generated OverPy code is fairly similar to Python with it's indentation, member variable calls and function syntax, this paper proposes a gradual method on this OverPy-syntaxed code, as it could have a similar performance as Hedy, where learning Python is made conceptually easier [Her20].

2.5 Gradual OverPy

OverPy itself is relatively intuitive. However, there are some functions that require many parameters to function. The function that displays simple text is called hudHeader. This function requires 7 parameters: what instances will see the text, the text itself, the positioning, sorting order, text color, reevaluation and spectator visibility. Most of these are irrelevant when trying to display text like "Hello world". This is where Gradual OverPy comes in, which can simplify the syntax of such complex functions and narrow it down to a simple function with 1 parameter, namely the text. Specifications on the grammar can be found in section 3.2.3

2.6 Relevance

The concept of the gradual language approach on Python by Hermans et al. is interesting by itself, but this concept could potentially be a frequently used teaching method when it comes to teaching a programming language. Applying such method on different languages has great learning potential. However, the effects of such method in a game environment or game-oriented method on gradual learning are still unknown. The result could be advantageous, like quicker understanding of concepts, or it could have drawbacks, like getting distracted quickly by the game.

3 Method

The research focuses on the incorporation of elements from the gradual method in a web application. The created web application is then used for qualitative research involving the participation of volunteers, who will be asked to test and provide their opinion on the web application.

3.1 Creating the web application

3.1.1 Requirements

As for technical requirements of the web application, they are as follows:

- The application should be accessible online through the internet browser
  - The application should be able to display HTML code
- The application should be able to compile OverPy code
  - The application should be able to run Javascript code (as the compilation is done with Javascript)
- The application should be able to use the Lark module from Python, as this will be the grammar parser
  - The application should be able to run Python code

When considering the requirements above, the choice was made to use the Python module Flask for this web application, as Flask allows for basic HTML/Javascript webpages and for running Python code in the backend. The webpage lay-out and OverPy compilation were done using HTML and Javascript, and the grammar parsing was done with the Python module Lark.

3.1.2 Flask

The main framework of the application uses Flask. Compared to the Python module http.server, Flask allows for running interactive Python code. In this web application, the parsing of the user's code with the module Lark is done in Python, hence the usage of Flask.

3.1.3 Lark parsing & grammar

Lark is a module that parses code in a given programmable grammar. In this web application, the grammar was written with the goal to simplify the OverPy syntax, cutting unnecessary elements from a given function if possible. Therefore, Lark will detect code with this specific grammar, and based on the functions it finds, will it return code in proper OverPy syntax. With this new simplified syntax, users need not to provide irrelevant elements.

3.2 The web application

3.2.1 End goal

The end goal of the web application is to simulate a jetpack system, comparable to the jetpack the character Pharah has, which includes a fuel tank, adding/removing fuel and vertical impulses. The pseudocode is as follows:

Algorithm 1 Simulated jetpack system

```
Define current player User
Define player variables Flying, Fuel
while Game is ongoing do
    if User is holding JUMP then
        User.Flying = true
    else                                        ▷ User is not holding JUMP
        User.Flying = false
    end if
    if User.Flying == true then
        if User.Fuel > 0 then
            User.Fuel = User.Fuel - 1
            Apply vertical impulse upwards
            Wait 10 ms
        else
            User.Fuel = 0                       ▷ This is to prevent Fuel going below 0
        end if
    else                                        ▷ Flying == false
        if User.Fuel < 100 then
            User.Fuel = User.Fuel + 1
            Wait 10 ms
        else
            User.Fuel = 100                     ▷ This is to prevent Fuel going above 100
        end if
    end if
end while
```

The reason that Flying is used when jumping instead of immediately checking the fuel, is to improve the clarity. When you are holding JUMP, you are flying, when you are flying, fuel is checked etc. This is by no means necessary and/or the optimal pseudocode for such system, but such optimized code is not required, as it might be more difficult to understand. In OverPy, it is possible to define all the code within one rule, but it is recommended to use multiple rules, just like using multiple functions in Python.

The eventual web application has the following structure:

- A main webpage, where (the purpose of) (Gradual) OverPy is introduced, the structure of the webpage is explained and how the code from the webpage can be transferred to the game to show the results (the latter one is done through a video).

- Stage 1, which contains 4 levels, each on a different webpage. Each level page is structured as follows:
  - The learning goal of the current level
  - The introduction and explanation of new elements
  - Example code for each introduced element
  - A inline editor to code programs with

3.2.2 The grammar of OverPy

OverPy has a syntax, which is pretty similar to Python. Some examples are indentation when defining if's, elses, while and for loops, member variables are called with a period and functions are called with round brackets. The following code blocks show how Python code looks like in OverPy.

```
Python3:
#In this example, we create a class named User for our event player. In OverPy,
#the code is running for every user and eventPlayer is predefined.
print("Hello world")
class User:
    def __init__(self):
        self.apples = 5
        self.pears = 2

    def changeValues(self, apples, pears):
        self.apples = apples
        self.pears = pears

eventPlayer = User()
eventPlayer.changeValues(5, 2)
if eventPlayer.apples > eventPlayer.pears:
    print("the eventPlayer has more apples than pears")
```

```
OverPy:
playervar 0 apples
playervar 1 pears

rule "beginCode":
    #The @Event defines that the code is running "within" every user.
    #This makes eventPlayer accessible
    @Event eachPlayer

    eventPlayer.apples = 5
    eventPlayer.pears = 2

```

```
rule "changeValues":
    @Event eachplayer

    #You need some way to trigger the code, so in this case,
    #we use the jump button
    if eventPlayer.isHoldingButton(Button.JUMP):
        eventPlayer.apples = 5
        eventPlayer.pears = 2

rule "printValues":
    @Event eachPlayer

    hudHeader(eventPlayer, "Hello world", HudPosition.LEFT, 0, Color.WHITE,
        HudReeval.VISIBILITY_AND_STRING, SpecVisibility.DEFAULT)
    if eventPlayer.apples > eventPlayer.pears:
        hudHeader(eventPlayer, "The eventPlayer has more apples than pears",
            HudPosition.LEFT, 0, Color.WHITE, HudReeval.VISIBILITY_AND_STRING,
            SpecVisibility.DEFAULT)
```

3.2.3 The grammar of Gradual OverPy

The grammar of all the 4 levels in Stage 1 is the same. This means that, for example, code from level 1 works in level 4 and vice versa. In Table 1 (see Section 7), the levels and the introduced concepts per level can be seen.

Noticable from the table is that most elements were left unchanged. This was done as those elements all contain necessary and relevant variables/elements, and a properly ordered introduction to such elements could suffice. The main changes in grammar were the hudHeader, the progressBarHud and applyImpulse. The hudHeader is supposed to be the print function in Python, the simple way to display text. The original OverPy syntax requires the user to provide many variables, which are not as relevant as the text itself. All variables like the position and color were removed and the simplified syntax only asks for text (and at level 3, the inclusion of (player) variables in the text is introduced). The back-end will fill in predefined code to match original OverPy, so that OverPy compilation succeeds with initially different syntax.

The hudHeader is detected by Lark with the following regular expressions:

```
INDENT INDENT* "hudHeader(" quoted_text_no_escape ")"
INDENT INDENT* "hudHeader(" textwithoutspaces BRACKETS? ")"
INDENT INDENT* "hudHeader(" ((textwithoutspaces BRACKETS? SPACE?
   quoted_text_no_escape) | (quoted_text_no_escape SPACE? textwithoutspaces
   BRACKETS?)) (quoted_text_no_escape? SPACE? textwithoutspaces? BRACKETS?)* ")"
```

The variables are defined as follows:

```
INDENT: "    "
BRACKETS: "()"
quoted_text_no_escape: /'([^']*)'/
textwithoutspaces: /([^\n *+-\/()]+\.?[^\n *+-\/()]+)/
```

The first two are fairly simple. Either it takes a string as input or a variable/function that provides a value. The third one is to make all combinations of strings and variables possible. It can handle the following syntax cases:

```
s = string
vf = variable or function (both function the same, as the function is
     textwithoutspaces but with brackets after)
s vf = string followed by a variable/function
s? = string can occur 0 or 1 times
s* = string can occur 0 or multiple times (any amount)

hudHeader(s)
hudHeader(s vf)
hudHeader(s vf s)
hudHeader(s vf s vf...)
hudHeader(s vf (s? vf?)*)
hudHeader(vf)
hudHeader(vf s)
hudHeader(vf s vf)
hudHeader(vf s vf s...)
hudHeader(vf s (s? vf?)*) (because (s? vf?)* = (vf? s?)*)
combined:
hudHeader( (s vf | vf s) (s? vf?)* )
```

Because the combined regular expression requires either a string and variable/function or a variable/function and a string, the cases with only a string or a variable/function had to be written out separately.
The proof for (s? vf?)* = (vf? s?)* is as follows:

```
(s? vf?)* = ((s? vf?)*)* = ((s? vf?)(s? vf?)(s? vf?)*)* = ((vf?)(s?)(s? vf?)*)*
  = ((vf?)(s?))* = (vf? s?)*
```

The progressBarHud and applyImpulse are detected as follows:

```
INDENT INDENT* "progressBarHud(" textwithoutcomma "," SPACE?
    (quoted_text_no_escape | textwithoutcomma) ")"
INDENT INDENT* eventplayer? "applyImpulse(" textwithoutcomma "," SPACE? text ")"

SPACE = " "
```

```
textwithoutcomma: /([^\n(),])+/
eventplayer = "eventplayer."
```

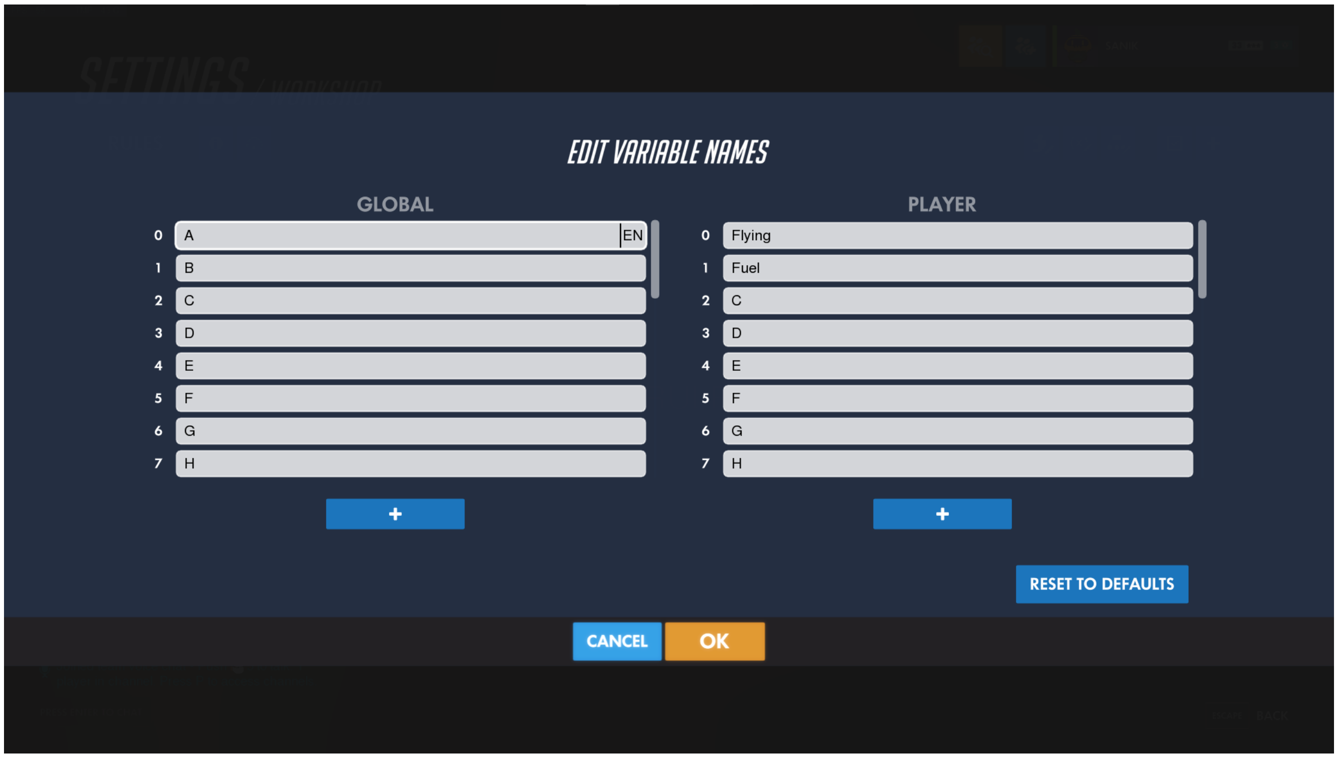
Any other code that is not detected by the regular expressions above will remain the same during OverPy compilation.

3.2.4 The levels of Gradual OverPy

The levels were structured in a way that new elements indirectly depend on previous elements. A simple example is the hudHeader in the first level, which displays text. This can be used to display the user input in the second level (which checks what buttons the user is pressing).

```
Gradual OverPy:
rule "displayText":
    hudHeader("Hello world")

rule "displayJump":
    @Event eachPlayer

    if eventPlayer.isHoldingButton(Button.JUMP):
        hudHeader("Jump")
```

In level 2, the @Event is introduced, which determines the ”location” of the rule. If combined with eachPlayer, it simply means that the rule is executed within every player that is present. With @Event eachPlayer, the eventPlayer and it’s member variables and functions can be called within the rule. The provided variables in this tutorial are the getCurrentHero() and isHoldingButton(). The getCurrentHero() function is fairly intuitive and easy to understand. The isHoldingButton() is a function that has many uses, including triggering when a rule is executed in combination with @Condition, which should be true, and only then is the rule executed. if a rule has multiple conditions, all those conditions should be satisfied before rule execution.
In level 3, variables are introduced. The standard variables are named after the capital letters of the alphabet (A, B, . . . , Z), but their names can be changed with a simple line above the rules. Depending on if the variable is a global or player variable should globalvar or playervar be used. The index indicates which variable’s name is changed (0 corresponds to A, 1 to B, 2 to C, etc.). This is not required to run proper programs, but it does allow for better code interpretation. The in-game overlay has a dedicated menu for changing variables (see Figure 6), which might improve the understanding of the explanation.



GLOBAL		PLAYER	
0	A	0	Flying
1	B	1	Fuel
2	C	2	C
3	D	3	D
4	E	4	E
5	F	5	F
6	G	6	G
7	H	7	H

Figure 6: Overwatch Workshop variable name change menu

Variables can be used to store integers, strings, booleans, etc., but they can also be displayed with hudHeader. The original OverPy syntax requires the variables to be indicated with 0, 1, etc. and the variables themselves should be put inside a format function. This is quite difficult to explain, so instead, the grammar was changed to make a distinct split between strings and variables with the double quotation marks.
The progressBarHud is also introduced, but with only 2 parameters: the value of the bar and the title. The maximum value of the bar is 100 by default. Examples are given of the bar being empty, completely filled and halfway filled.
In the final level the applyImpulse is introduced, which simply applies a (vertical) impulse to the player. This was introduced as the last element for the jetpack system, as the other levels contain the base for the whole system itself. Most of the introduced elements are used for creating the jetpack. Some elements are introduced because of the relevance of that current level. In level 2, destroyHudText(getLastCreatedText()) is shown as an extra element, which removes the last added hudHeader. Even though this element is not used in the final jetpack system, it can be used to test programs in level 2.
Something that should be noted is that, because of the grammar changes compared to original OverPy, if regular OverPy code is used to attempt code compilation, it will result in errors. Original OverPy code can be compiled with the Demo [OvPb], provided by Zezombye himself.

3.2.5 Comparison between Gradual OverPy, Hedy and the in-game overlay

Because Gradual OverPy is heavily based on Hedy, the interface is somewhat similar. Similarities are the dedicated editor and the possibility to insert example code. However, there are many

	Hedy	Gradual OverPy	In-game overlay
Based on	Python	OverPy	Workshop language
Introduction to new elements	Independent on other functions	Dependent on other functions	None
	Small descriptions	Large descriptions	Small descriptions
Running code	Within integrated console	In game	In game
Error handling	Simplified error messages	Regular error messages	Regular error messages

differences because of the structure of OverPy's code. One major difference is the explanation of functions. Hedy has small descriptions, whereas Gradual OverPy has large explanations, which the intention was to prevent abstraction. Another difference is the result of running code. As Hedy is based on Python, the result is (in early levels) in text form, which is directly visible from the web application. Gradual OverPy on the other hand generates workshop code for the user to import themselves in the game. This is an extra step required for the user to see results, but the result can be interactive and more than just textual output. Another quality of life feature in Hedy is the simplification of error messages, one example being provided suggestions in case of syntax errors. Gradual OverPy does not have such feature.

Gradual OverPy has quite some similarities with the in-game overlay, including the same result when running code. The major difference is that the overlay has no integrated tutorial, whereas Gradual OverPy is a tutorial in itself and it introduces elements based on previously introduced elements per level. Even though the overlay provides small descriptions for each function when hovering over an element with the cursor, these could still lead to abstraction, because of incomplete explanations. The error messages on the other hand provide direct feedback and refer to specific elements, whereas Gradual OverPy has vague error messages dedicated to OverPy instead of Gradual OverPy.

3.3 Testing the web application

The web application was tested using interviews. Interviewees went through the 4 levels and qualitative data was then collected. These interviewees' programming skills range from inexperienced to (somewhat) experienced. This will roughly show how effective and desirable this method is depending on different skill ranges.

The reason why qualitative research was done instead of quantitative research is that the web application does require the user to commit to learning a fully new programming language. This requires a level of commitment and covers a range of details which is difficult to adapt to online quantitative testing (especially in the current pandemic). If during the interview the interviewee thinks that the web application is boring, they are not forced to continue, which still provides better information than expecting large amounts of testing online, which has the risk of ending up with small/no results.

The interviewees were told the following:

- ”Hi, thanks for your time. First off, am I allowed to record this think-aloud interview? This is purely for citing purposes and there are no means of harm.”
- ”Secondly: It would be ideal to revise the whole first stage, but if you think halfway through it gets boring or you do not understand anything, feel free to let me know and we will go to the questions right away.”
- ”If you read something and don't quite understand what it means, you can ask me and I will try to explain how it works.”
- ”Now before you can start with the first level, I will ask you some quick questions about you and your experience in programming.”
  - ”What is your age?”
  - ”What is your gender?”
  - ”How experienced are you with programming? And what languages do you know?”
  - ”How interested are you in programming?”

After this, the interviewees started with the first level. During the interview, the interviewee could ask questions about the project or the content if they did not understand how certain elements function. At the end of each level, a code comprehension test was done, where they were asked to go through the example code provided with the ”Insert example” button, and explain what the code does. The initial protocol had them write code themselves, but as (Gradual) OverPy could need many rules for simple functionality, which can be a time-consuming process, it was decided that explaining the code would be sufficient enough for the code comprehension test.
After the interviewees went through all the levels (or after they ended it early due to boredom/lack of interest), they were asked the following questions:

- ”How did it go? Did it go smoothly or did you have to reread the descriptions? How far did they come?”
- ”What are some things that you liked and what parts did you not like (and how do you think you could improve it)?”
  - ”Did you like the integrated editor? Do you think the explanations were clear? Should there be videos per level instead to keep the users attention? Should there be (more) challenges?”
- ”Do you think this concept would be interesting to other players, beginning programmers, players that want to explore the workshop?”
- ”Do you think you have mastered the elements from the levels? And do you think you can master more elements with this approach?”
- ”Have you tried the in-game overlay? Do you think this is a good replacement? Or only for people that want to learn programming?”

4 Results

During development, a first prototype was shared online within a small social network. This resulted in only first impressions, and no proper testing. Barbero also revised this prototype and provided minor improvements.
Screenshots of the developed web application (Figures 7, 8, 9, 10, 11 and 12) can be found in Section 7.

As for the interviews, the interviewees were interviewed online through a voice call in the communication application called Discord [Inc15]. The results of the interviews can be seen in Table 2 at Section 7.

To summarize the results:
The interviewees managed to finish all the levels, some with more ease than others. The first interviewee found the explanations clear with the provided examples, whilst the other two had trouble understanding the introduced concepts, one because of the complexity, the other because of the incompleteness. Noticeable is that all interviewees liked the integrated editor. The interviewee with no experience liked the "insert example" button, as it provides a template, however, this template becomes more complicated at later levels.

As for direct feedback on the web application, the two main potential improvements they thought might help were:

1. Adding videos for the introduction of an element

2. Adding small challenges per level

Their opinion on the concept is positive in general. It could be a great replacement for learning the original workshop syntax, especially for beginning programmers, as they think the workshop code is quite messy compared to Gradual OverPy.

As for mastering elements, the two interviewees who are (somewhat) experienced in programming could master more elements if this approach were to be refined in the future, as they did not master the elements through the testing.

Out of the interviewees, none of them have tried the in-game programming overlay. After showing how the overlay works, the first interviewee thought it was easier than Gradual OverPy. The second interviewee thought it looked easier, but having that many predefined elements to experiment with would probably be as hard as learning Gradual OverPy. As the overlay is just an editor in its core, having a dedicated tutorial is better, as it guides the programmer through the large amount of elements that exist. The third interviewee thought that the overlay was ambiguous and that Gradual OverPy would be better to use, even if people want to quickly experiment.

5 Discussion

As mentioned in Section 3, the provided explanations for the introduced elements in the web application did not suffice for all the interviewees. This was due to lengthy text blocks which attempt to explain how, where and when certain elements have to be used. This could have caused the explanation to be too long for people that want a quick explanation on the usage and it would be too short/informal for programmers that are used to having proper documentation.

The initial target group was the group of Overwatch players who have small to no programming knowledge and are interested in learning both programming and the Overwatch workshop, but looking at the results, players who already know basic/complex programming could also benefit from Gradual OverPy, if they are interested in workshop programming. To prevent the exclusion of a specific target group, the large explanations should be replaced with both short simple usage explanations for beginners and complete documentation per function/element for experienced programmers. Most of the documentation is similar to the documentation from OverPy. The main differences lay in the functions/elements with changed syntax, so including documentation per stage on how the syntax of functions has changed in that stage might already be sufficient.

When looking at the contents of the web application, even though the first stage covers the whole of the jetpack system, this is still just a proof of concept, as the web application only contains one stage with four levels. The only changes to the actual grammar were the hudHeader and progressBarHud functions. The gradual change to OverPy has not been implemented yet because of this.

Two problems which have not been discussed are the website layout and the organization of code of the web application.
The current layout of the levels is very basic and is minimalistic: It is bland and has almost no CSS. This was done as the focus of this project lays in the functionality rather than the looks of the web application. The functionality of the application is present, but changing one website element from all levels has to be done manually. Making more use of blocks for repetitive usage would be a great improvement for future development.

6 Conclusions and future work

Gradual OverPy, the concept of the gradual learning method of OverPy, has shown that a gradual method on an lesser-known language could be of interest. As seen in the results, people with different programming experiences can show similarities and differences in their opinion, but the important part is that these opinions are brought out and should be considered in cases of future development, as Gradual OverPy is currently still a proof of concept. To answer the research question ”Does a gradual language system help Overwatch players develop custom workshops?”, the generosity and interest of the players should be considered. Depending on what a player desires to learn/do, can Gradual OverPy be a tool to gain knowledge both about the elements of the game and programming. Therefore, the answer to the research question is: ”Yes, but their learning potential mostly depends on the generosity and interest of the player.” This conclusion can be backed by Section 2, where syntax requires user insight to get an understanding of it. A gradual method on the programming language of a game can work for players that are interested, independent of their insight on syntax.

In the case this project would be continued, the following (mentioned) potential improvements could be applied:

- Adding video footage for each introduced element
- Changing the blocks of text with explanation to:
  - Simple usage explanation
  - Complete fully explained documentation
- Improve website layout
- Better code organization

A bigger research could be if a gradual method on OverPy would teach beginning programmers to program could be researched by implementing the gradual method itself and completely covering all the syntax changes such that the resulting code is original OverPy instead of a simplified version of OverPy.

References

[CM08] Ioana Chan Mow. *Issues and Difficulties in Teaching Novice Computer Programming*, pages 199–204. 08 2008.

[ENS06] Magy Seif El-Nasr and Brian K. Smith. Learning through game modding. *Comput. Entertain.*, 4(1):7–es, January 2006.

[Ent16] Blizzard Entertainment. Overwatch. PC/PlayStation 4/Xbox One/Nintendo Switch, 2016.

[Her20] Felienne Hermans. Hedy: A gradual language for programming education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 259–270, New York, NY, USA, 2020. Association for Computing Machinery.

[Inc15] Discord Inc. Discord. PC/iOS/Android/Linux, 2015.

[OvPa] OverPy v6.0. `https://github.com/Zezombye/overpy`. Accessed: 2021-3-30.

[OvPb] OverPy v6.0 language demo by zezombye. `https://zezombye.github.io/overpy/demo`. Accessed: 2021-3-30.

[VC12] Hidden Path Entertainment Valve Corporation. Counter strike: Global offensive. PC, 2012.

7 Appendix

Level	**Original OverPy code**
Level 1	rule ”myRule”:
Simplified syntax	*unchanged*
	hudHeader(eventPlayer, ”Jump”, HudPosition.LEFT, 0, Color.WHITE, HudReeval.VISIBILITY_AND_STRING, SpecVisibility.DEFAULT)
Simplified syntax	hudHeader(”Jump”)
Level 2	@Event eachPlayer
Simplified syntax	*unchanged*
	@Condition eventPlayer.getCurrentHero() != Hero.PHARAH @Condition eventPlayer.isHoldingButton(Button.JUMP) == true
Simplified syntax	*unchanged*
	destroyHudText(getLastCreatedText())
Simplified syntax	*unchanged*
Level 3	globalvar apples 0
Simplified syntax	*unchanged*
	playervar apples 0
Simplified syntax	*unchanged*

	apples = 5
	pears = 7
	eventPlayer.apples = 5
	eventPlayer.Flying = true
	eventPlayer.Text = ”Sample text”

Simplified syntax	*unchanged*
	hudHeader(getAllPlayers(), ”There are {0} apples and {1} pears”.format(apples.pears), HudPosition.LEFT, 0, Color.WHITE, HudReeval.VISIBILITY_AND_STRING, SpecVisibility.DEFAULT) hudHeader(eventPlayer, ”You have {0} apples and {1} pears”.format(eventPlayer.apples, eventPlayer.pears), HudPosition.LEFT, 0, Color.WHITE, HudReeval.VISIBILITY_AND_STRING, SpecVisibility.DEFAULT)

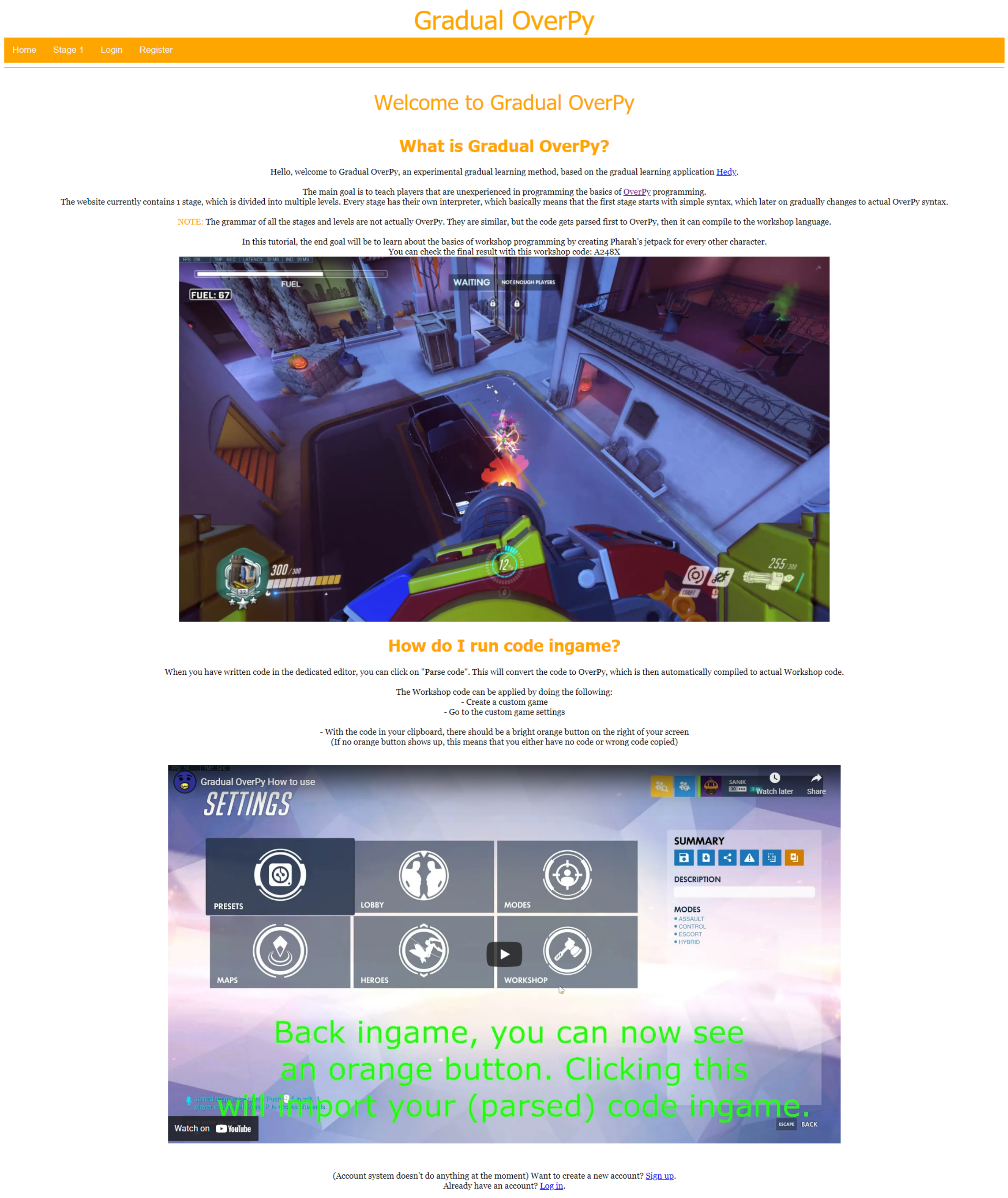
Simplified syntax	hudHeader(”There are ” apples ” apples and ” pears ” pears.”)
	hudHeader(”You have ” eventPlayer.apples ” apples and ”eventPlayer.pears ” pears.”)

	eventPlayer.sum = 2 + 30
Simplified syntax	*unchanged*
	progressBarHud(eventPlayer, eventPlayer.Fuel, ”Fuel”, HudPosition.LEFT, 0, Color.WHITE, Color.WHITE, ProgressHudReeval.VISIBILITY_VALUES_AND_COLOR, SpecVisibility.DEFAULT)
Simplified syntax	progressBarHud(eventPlayer.Fuel, ”Fuel”)
	if RULE_CONDITION: goto RULE_START
Simplified syntax	*unchanged*
Level 4	eventPlayer.applyImpulse(Vector.UP, 0.3, Relativity.TO_PLAYER, Impulse.INCORPORATE_CONTRARY_MOTION)
Simplified syntax	eventPlayer.applyImpulse(Vector.UP, 0.3)

Table 1: Grammar of (Gradual) OverPy of each element introduced in each level

	Interviewee 1	Interviewee 2	Interviewee 3
Age, gender, programming experience	19, male, knows Java from high school	22, male, no experience	21, male, experienced in C++, Python, Javascript (2 years), PHP (3+ years)
Interest in learning a programming language	Not really	Generally no, but seems fun to be able to program	Very interested
Progress and smoothness of the interview	Finished all levels, went quite well, example code was easy to understand	Finished all levels with assistance. First few levels were doable, but further levels were difficult without verbal explanations. Importing and changing code makes code comprehension easier.	Finished all levels, went pretty smooth because of programming experience
Liked/Disliked parts	Liked: -Integrated editor: good to check if the code could run -Many examples: makes the explanations clearer -Easy to read Disliked: None	Liked: -Insert example button -Editor is amazing Disliked: -Insert example button at later levels: quite a lot of code to understand. -Explanations read but not fully comprehensive.	Liked: -Integrated editor Disliked: -Incomplete explanations: not everything was fully explained, some parts in the explanations were missing.
Potential improvements	Some people prefer text and examples, some prefer videos. Doing both would be the best solution.	During the interview, the incomprehension was solved through verbal explanations. It could be solved through reoccurance of first few elements at later levels through small challenges.	Adding videos is definitely recommended, Having small challenges per level could help code comprehension overall.
Opinion on concept	Not sure how easy it is for people that are beginning to learn programming, but for experienced programmers quite understandable. If someone is into Overwatch, it could open a whole new world. They could start coding their own gamemodes and experiment with elements. It really depends on how that person wants to get into programming. It is generally a good idea, because the original workshop code is quite a mess.	Thinks the concept is interesting, as a decent amount of players would probably want to try things out but don't know where to start. It would be more interesting to beginning programmers. Half a year of experience would be ideal. Great introduction to the workshop Gradual OverPy is way easier to understand than the workshop code. With Gradual OverPy and the workshop code side by side, elements in the workshop code can be spotted.	Not sure about players, as he is not involved in the Overwatch scene. It would probably be interesting for beginning programmers.
Elements mastered/ able to master elements with this approach	Would go back to this website to check the syntax if he wanted to code something by himself. Think he would probably be able to master more elements with this approach.	Has not really mastered the elements, but if he got constant vocal feedback, he could probably create some things after a few days.	Has not completely mastered the elements, because there is no complete documentation/ explanation. Could master more elements with this approach
Ingame overlay, Gradual OverPy as replacement for the overlay	Has not tried the ingame overlay. After demonstration: Ingame overlay would be easier, but if someone wants to learn programming, Gradual OverPy would be better	Has not tried the ingame overlay. After demonstration: At first glance, the overlay seems to be easier to code, especially doing small experiments. Even though elements are predefined in a list, there are quite a lot of elements, which is probably as difficult as learning syntax. In this case, having a dedicated tutorial is better than the overlay by itself.	Has not tried the ingame overlay. After demonstration: Did not really understand the overlay. Gradual OverPy is a good replacement for it. Gradual OverPy would also be handy for people that want to quickly put something together.

Table 2: Results of the interviews



Gradual OverPy

Home Stage 1 Login Register

Welcome to Gradual OverPy

What is Gradual OverPy?

Hello, welcome to Gradual OverPy, an experimental gradual learning method, based on the gradual learning application Hedy.

The main goal is to teach players that are unexperienced in programming the basics of OverPy programming.
The website currently contains 1 stage, which is divided into multiple levels. Every stage has their own interpreter, which basically means that the first stage starts with simple syntax, which later on gradually changes to actual OverPy syntax.

NOTE: The grammar of all the stages and levels are not actually OverPy. They are similar, but the code gets parsed first to OverPy, then it can compile to the workshop language.

In this tutorial, the end goal will be to learn about the basics of workshop programming by creating Pharah's jetpack for every other character.
You can check the final result with this workshop code: A248X

How do I run code ingame?

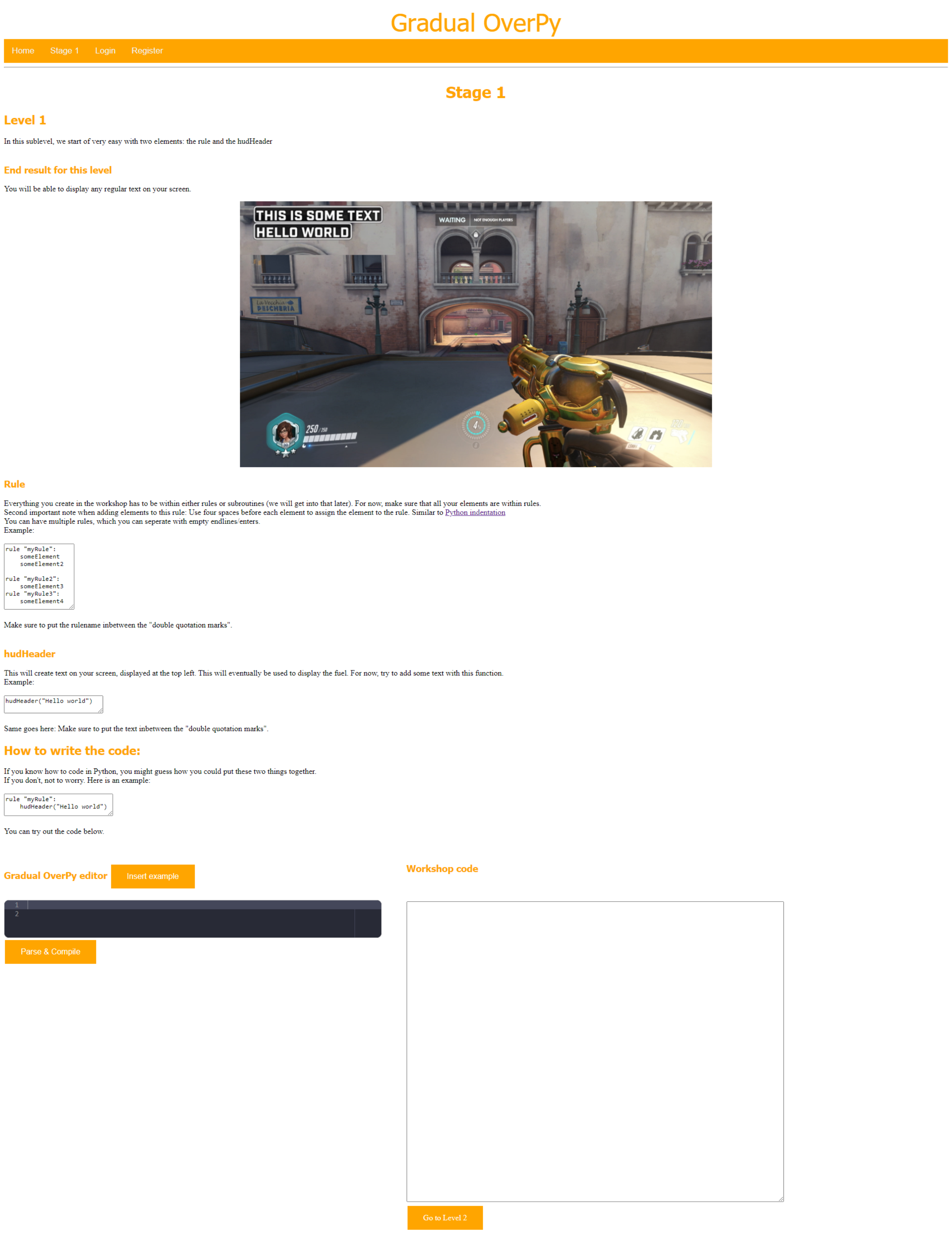
When you have written code in the dedicated editor, you can click on "Parse code". This will convert the code to OverPy, which is then automatically compiled to actual Workshop code.

The Workshop code can be applied by doing the following:
- Create a custom game
- Go to the custom game settings

- With the code in your clipboard, there should be a bright orange button on the right of your screen
(If no orange button shows up, this means that you either have no code or wrong code copied)

(Account system doesn't do anything at the moment) Want to create a new account? Sign up.
Already have an account? Log in.

Figure 7: Gradual OverPy main page



Gradual OverPy

Home Stage 1 Login Register

Stage 1

Level 1

In this sublevel, we start of very easy with two elements: the rule and the hudHeader

End result for this level

You will be able to display any regular text on your screen.

Rule

Everything you create in the workshop has to be within either rules or subroutines (we will get into that later). For now, make sure that all your elements are within rules.
Second important note when adding elements to this rule: Use four spaces before each element to assign the element to the rule. Similar to Python indentation
You can have multiple rules, which you can seperate with empty endlines/enters.
Example:

```
rule "myRule":
    someElement
    someElement2

rule "myRule2":
    someElement3
rule "myRule3":
    someElement4
```

Make sure to put the rulename inbetween the "double quotation marks".

hudHeader

This will create text on your screen, displayed at the top left. This will eventually be used to display the fuel. For now, try to add some text with this function.
Example:

```
hudHeader("Hello world")
```

Same goes here: Make sure to put the text inbetween the "double quotation marks".

How to write the code:

If you know how to code in Python, you might guess how you could put these two things together.
If you don't, not to worry. Here is an example:

```
rule "myRule":
    hudHeader("Hello world")
```

You can try out the code below.

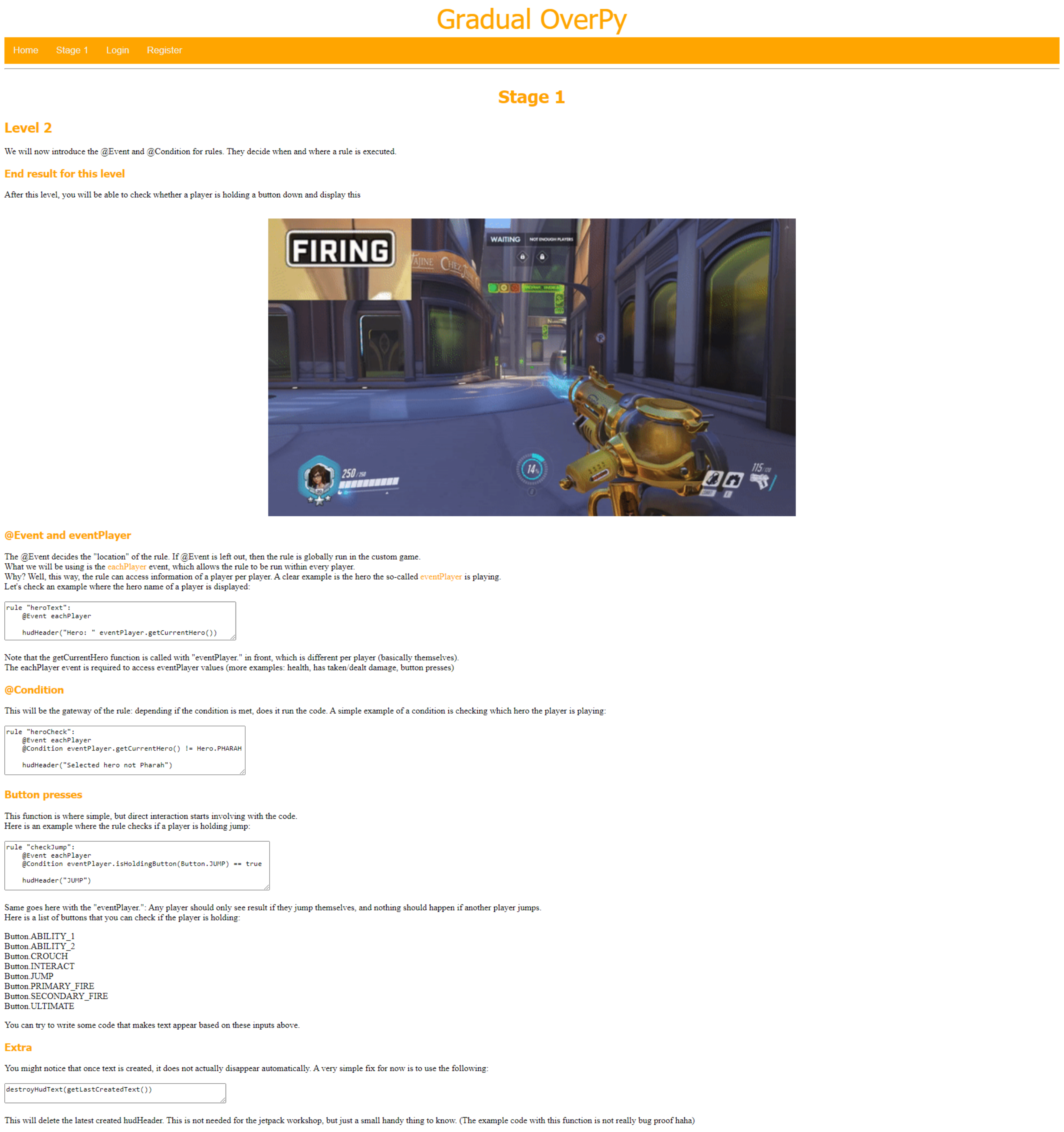
Gradual OverPy editor Insert example

Workshop code

Parse & Compile

Go to Level 2

Figure 8: Gradual OverPy Stage 1 Level 1



Gradual OverPy

Home Stage 1 Login Register

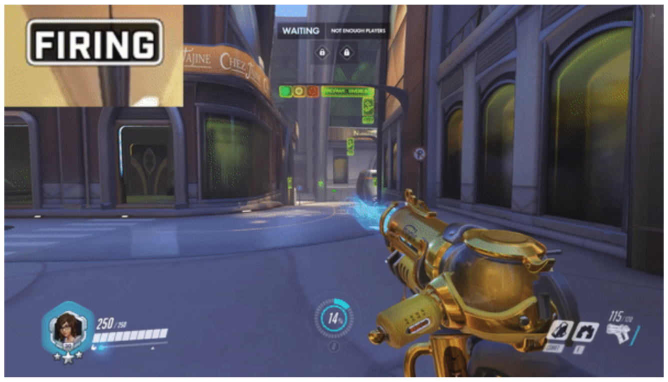
Stage 1

Level 2

We will now introduce the @Event and @Condition for rules. They decide when and where a rule is executed.

End result for this level

After this level, you will be able to check whether a player is holding a button down and display this



@Event and eventPlayer

The @Event decides the "location" of the rule. If @Event is left out, then the rule is globally run in the custom game.
What we will be using is the eachPlayer event, which allows the rule to be run within every player.
Why? Well, this way, the rule can access information of a player per player. A clear example is the hero the so-called eventPlayer is playing.
Let's check an example where the hero name of a player is displayed:

```
rule "heroText":
    @Event eachPlayer

    hudHeader("Hero: " eventPlayer.getCurrentHero())
```

Note that the getCurrentHero function is called with "eventPlayer." in front, which is different per player (basically themselves).
The eachPlayer event is required to access eventPlayer values (more examples: health, has taken/dealt damage, button presses)

@Condition

This will be the gateway of the rule: depending if the condition is met, does it run the code. A simple example of a condition is checking which hero the player is playing:

```
rule "heroCheck":
    @Event eachPlayer
    @Condition eventPlayer.getCurrentHero() != Hero.PHARAH

    hudHeader("Selected hero not Pharah")
```

Button presses

This function is where simple, but direct interaction starts involving with the code.
Here is an example where the rule checks if a player is holding jump:

```
rule "checkJump":
    @Event eachPlayer
    @Condition eventPlayer.isHoldingButton(Button.JUMP) == true

    hudHeader("JUMP")
```

Same goes here with the "eventPlayer.": Any player should only see result if they jump themselves, and nothing should happen if another player jumps.
Here is a list of buttons that you can check if the player is holding:

Button.ABILITY_1
Button.ABILITY_2
Button.CROUCH
Button.INTERACT
Button.JUMP
Button.PRIMARY_FIRE
Button.SECONDARY_FIRE
Button.ULTIMATE

You can try to write some code that makes text appear based on these inputs above.

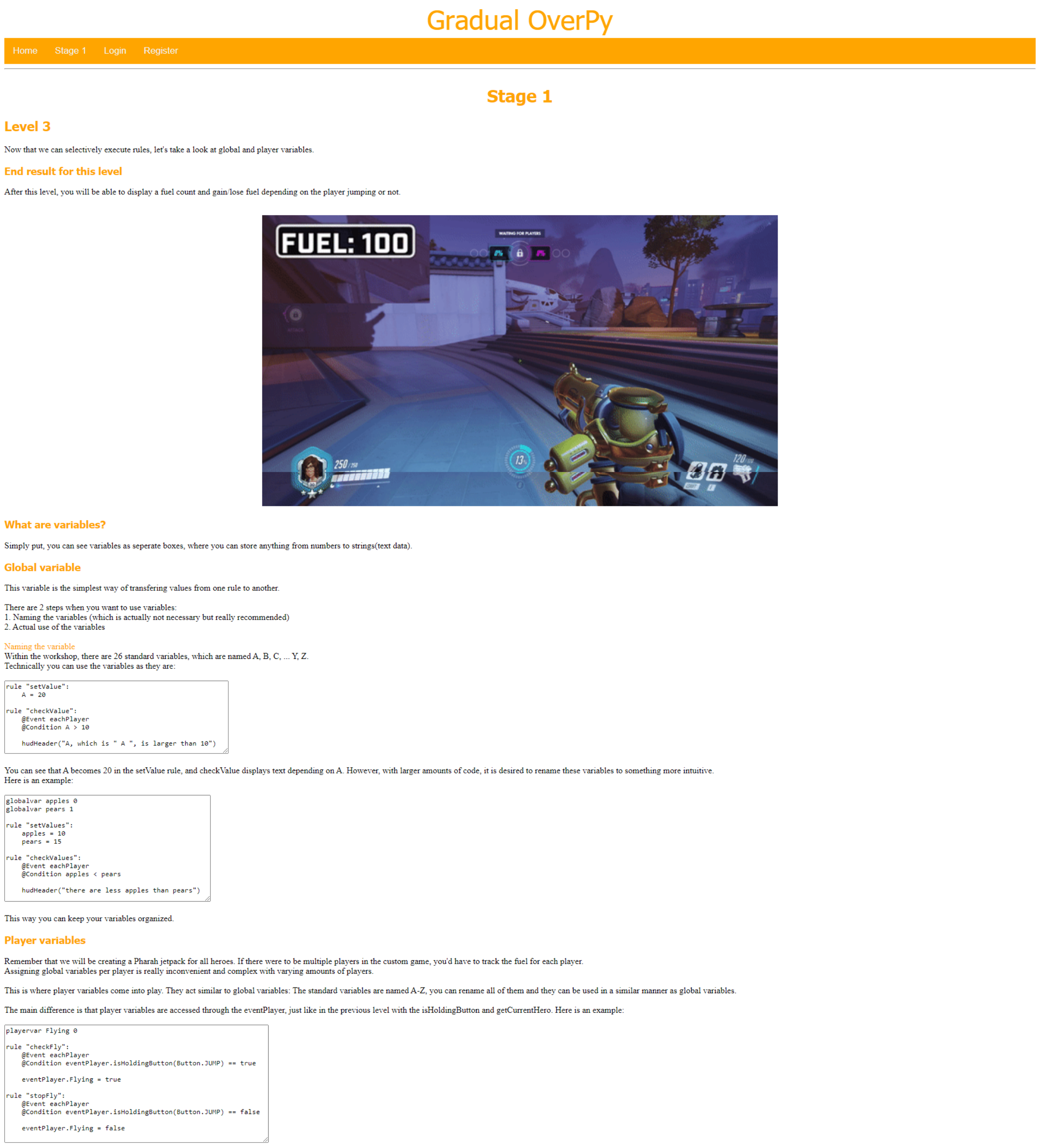
Extra

You might notice that once text is created, it does not actually disappear automatically. A very simple fix for now is to use the following:

```
destroyHudText(getLastCreatedText())
```

This will delete the latest created hudHeader. This is not needed for the jetpack workshop, but just a small handy thing to know. (The example code with this function is not really bug proof haha)

Figure 9: Gradual OverPy Stage 1 Level 2



Gradual OverPy

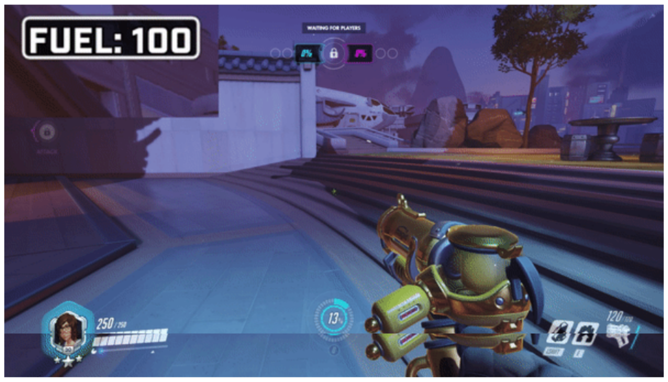
Stage 1

Level 3

Now that we can selectively execute rules, let's take a look at global and player variables.

End result for this level

After this level, you will be able to display a fuel count and gain/lose fuel depending on the player jumping or not.



What are variables?

Simply put, you can see variables as seperate boxes, where you can store anything from numbers to strings(text data).

Global variable

This variable is the simplest way of transfering values from one rule to another.

There are 2 steps when you want to use variables:
1. Naming the variables (which is actually not necessary but really recommended)
2. Actual use of the variables

Naming the variable
Within the workshop, there are 26 standard variables, which are named A, B, C, ... Y, Z.
Technically you can use the variables as they are:

```
rule "setValue":
    A = 20

rule "checkValue":
    @Event eachPlayer
    @Condition A > 10

    hudHeader("A, which is " A ", is larger than 10")
```

You can see that A becomes 20 in the setValue rule, and checkValue displays text depending on A. However, with larger amounts of code, it is desired to rename these variables to something more intuitive.
Here is an example:

```
globalvar apples 0
globalvar pears 1

rule "setValues":
    apples = 10
    pears = 15

rule "checkValues":
    @Event eachPlayer
    @Condition apples < pears

    hudHeader("there are less apples than pears")
```

This way you can keep your variables organized.

Player variables

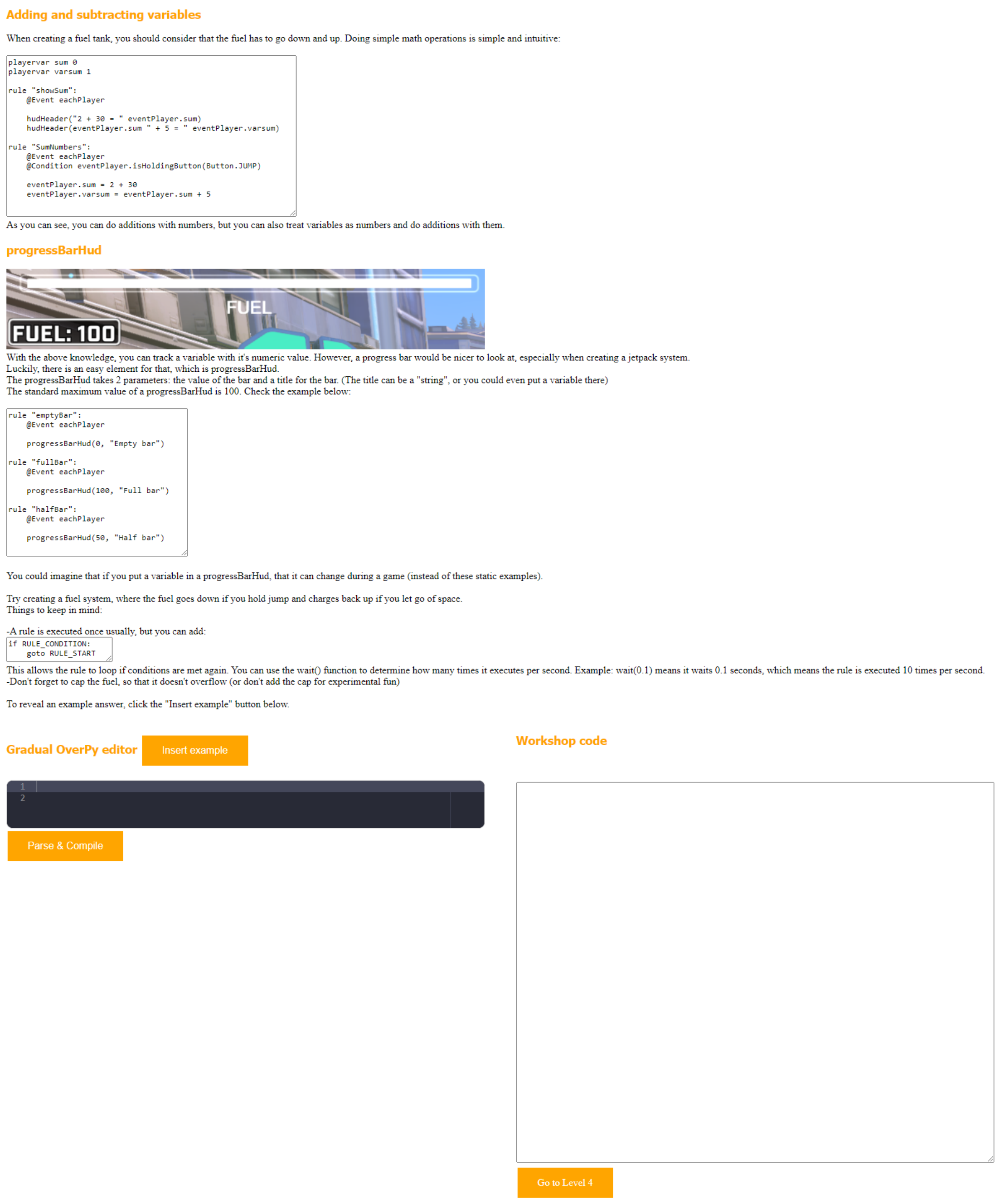
Remember that we will be creating a Pharah jetpack for all heroes. If there were to be multiple players in the custom game, you'd have to track the fuel for each player.
Assigning global variables per player is really inconvenient and complex with varying amounts of players.

This is where player variables come into play. They act similar to global variables: The standard variables are named A-Z, you can rename all of them and they can be used in a similar manner as global variables.

The main difference is that player variables are accessed through the eventPlayer, just like in the previous level with the isHoldingButton and getCurrentHero. Here is an example:

```
playervar Flying 0

rule "checkFly":
    @Event eachPlayer
    @Condition eventPlayer.isHoldingButton(Button.JUMP) == true

    eventPlayer.Flying = true

rule "stopFly":
    @Event eachPlayer
    @Condition eventPlayer.isHoldingButton(Button.JUMP) == false

    eventPlayer.Flying = false
```

Figure 10: Gradual OverPy Stage 1 Level 3 part 1



Adding and subtracting variables

When creating a fuel tank, you should consider that the fuel has to go down and up. Doing simple math operations is simple and intuitive:

```
playervar sum 0
playervar varsum 1

rule "showSum":
    @Event eachPlayer

    hudHeader("2 + 30 = " eventPlayer.sum)
    hudHeader(eventPlayer.sum " + 5 = " eventPlayer.varsum)

rule "SumNumbers":
    @Event eachPlayer
    @Condition eventPlayer.isHoldingButton(Button.JUMP)

    eventPlayer.sum = 2 + 30
    eventPlayer.varsum = eventPlayer.sum + 5
```

As you can see, you can do additions with numbers, but you can also treat variables as numbers and do additions with them.

progressBarHud

With the above knowledge, you can track a variable with it's numeric value. However, a progress bar would be nicer to look at, especially when creating a jetpack system.
Luckily, there is an easy element for that, which is progressBarHud.
The progressBarHud takes 2 parameters: the value of the bar and a title for the bar. (The title can be a "string", or you could even put a variable there)
The standard maximum value of a progressBarHud is 100. Check the example below:

```
rule "emptyBar":
    @Event eachPlayer

    progressBarHud(0, "Empty bar")

rule "fullBar":
    @Event eachPlayer

    progressBarHud(100, "Full bar")

rule "halfBar":
    @Event eachPlayer

    progressBarHud(50, "Half bar")
```

You could imagine that if you put a variable in a progressBarHud, that it can change during a game (instead of these static examples).

Try creating a fuel system, where the fuel goes down if you hold jump and charges back up if you let go of space.
Things to keep in mind:

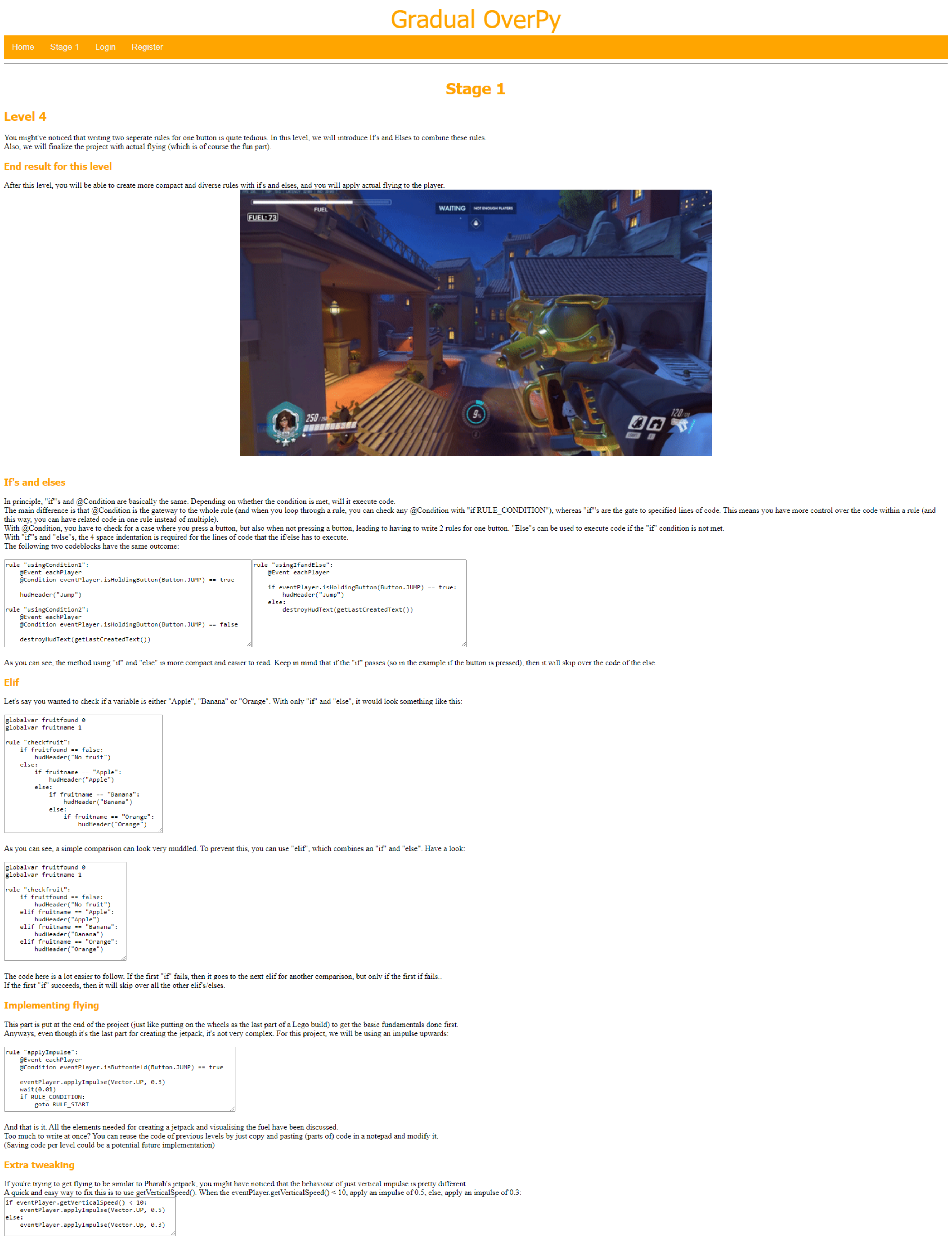
-A rule is executed once usually, but you can add:

```
if RULE_CONDITION:
    goto RULE_START
```

This allows the rule to loop if conditions are met again. You can use the wait() function to determine how many times it executes per second. Example: wait(0.1) means it waits 0.1 seconds, which means the rule is executed 10 times per second.
-Don't forget to cap the fuel, so that it doesn't overflow (or don't add the cap for experimental fun)

To reveal an example answer, click the "Insert example" button below.

Figure 11: Gradual OverPy Stage 1 Level 3 part 2



Gradual OverPy

Home Stage 1 Login Register

Stage 1

Level 4

You might've noticed that writing two seperate rules for one button is quite tedious. In this level, we will introduce If's and Elses to combine these rules.
Also, we will finalize the project with actual flying (which is of course the fun part).

End result for this level

After this level, you will be able to create more compact and diverse rules with if's and elses, and you will apply actual flying to the player.

If's and elses

In principle, "if"'s and @Condition are basically the same. Depending on whether the condition is met, will it execute code.
The main difference is that @Condition is the gateway to the whole rule (and when you loop through a rule, you can check any @Condition with "if RULE_CONDITION"), whereas "if"'s are the gate to specified lines of code. This means you have more control over the code within a rule (and this way, you can have related code in one rule instead of multiple).
With @Condition, you have to check for a case where you press a button, but also when not pressing a button, leading to having to write 2 rules for one button. "Else"s can be used to execute code if the "if" condition is not met.
With "if"'s and "else"s, the 4 space indentation is required for the lines of code that the if/else has to execute.
The following two codeblocks have the same outcome:

```
rule "usingCondition1":
    @Event eachPlayer
    @Condition eventPlayer.isHoldingButton(Button.JUMP) == true

    hudHeader("Jump")

rule "usingCondition2":
    @Event eachPlayer
    @Condition eventPlayer.isHoldingButton(Button.JUMP) == false

    destroyHudText(getLastCreatedText())
```

```
rule "usingIfandElse":
    @Event eachPlayer

    if eventPlayer.isHoldingButton(Button.JUMP) == true:
        hudHeader("Jump")
    else:
        destroyHudText(getLastCreatedText())
```

As you can see, the method using "if" and "else" is more compact and easier to read. Keep in mind that if the "if" passes (so in the example if the button is pressed), then it will skip over the code of the else.

Elif

Let's say you wanted to check if a variable is either "Apple", "Banana" or "Orange". With only "if" and "else", it would look something like this:

```
globalvar fruitfound 0
globalvar fruitname 1

rule "checkfruit":
    if fruitfound == false:
        hudHeader("No fruit")
    else:
        if fruitname == "Apple":
            hudHeader("Apple")
        else:
            if fruitname == "Banana":
                hudHeader("Banana")
            else:
                if fruitname == "Orange":
                    hudHeader("Orange")
```

As you can see, a simple comparison can look very muddled. To prevent this, you can use "elif", which combines an "if" and "else". Have a look:

```
globalvar fruitfound 0
globalvar fruitname 1

rule "checkfruit":
    if fruitfound == false:
        hudHeader("No fruit")
    elif fruitname == "Apple":
        hudHeader("Apple")
    elif fruitname == "Banana":
        hudHeader("Banana")
    elif fruitname == "Orange":
        hudHeader("Orange")
```

The code here is a lot easier to follow. If the first "if" fails, then it goes to the next elif for another comparison, but only if the first if fails..
If the first "if" succeeds, then it will skip over all the other elif's/elses.

Implementing flying

This part is put at the end of the project (just like putting on the wheels as the last part of a Lego build) to get the basic fundamentals done first.
Anyways, even though it's the last part for creating the jetpack, it's not very complex. For this project, we will be using an impulse upwards:

```
rule "applyImpulse":
    @Event eachPlayer
    @Condition eventPlayer.isButtonHeld(Button.JUMP) == true

    eventPlayer.applyImpulse(Vector.UP, 0.3)
    wait(0.01)
    if RULE_CONDITION:
        goto RULE_START
```

And that is it. All the elements needed for creating a jetpack and visualising the fuel have been discussed.
Too much to write at once? You can reuse the code of previous levels by just copy and pasting (parts of) code in a notepad and modify it.
(Saving code per level could be a potential future implementation)

Extra tweaking

If you're trying to get flying to be similar to Pharah's jetpack, you might have noticed that the behaviour of just vertical impulse is pretty different.
A quick and easy way to fix this is to use getVerticalSpeed(). When the eventPlayer.getVerticalSpeed() < 10, apply an impulse of 0.5, else, apply an impulse of 0.3:

```
if eventPlayer.getVerticalSpeed() < 10:
    eventPlayer.applyImpulse(Vector.UP, 0.5)
else:
    eventPlayer.applyImpulse(Vector.Up, 0.3)
```

Figure 12: Gradual OverPy Stage 1 Level 4