Dimension Surfer Game

Computing Project Documentation

Jakub Bartosz Dranczewski

Contents

Project Definition and Analysis

Project Definition

In this project, I am going to use Python and one of its libraries, pygame, to create a platform game. The player will be able to move left and right and jump, navigating obstacles and trying to get to the end of the level, as it would be expected in a classical platform game. The game will also have another unique mechanism of steering - manipulation of the third dimension. Through that I expect to introduce much more variety into the normal platform game model, as well as help the end user understand the complex concept of higher dimensions.

Suppose that the game's character is a two-dimensional being and as such can only see a two-dimensional world. Now if we placed this character in a three-dimensional world he wouldn't be able to comprehend it with his set of senses, so he wouldn't see the eternity of it, just a two-dimensional cross-section.

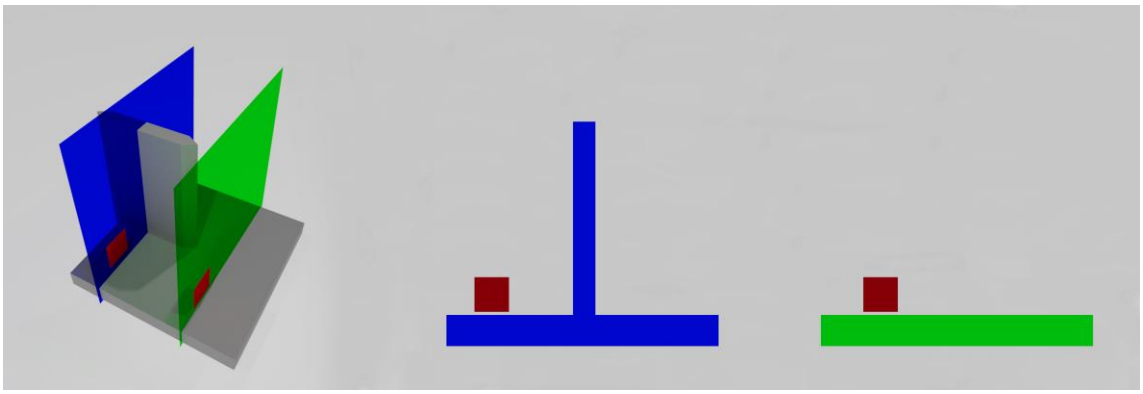


Figure 1: The cross-section idea: when the player chooses the blue cross-section, he sees a level with an obstacle in the middle. He can then switch to the green cross-section and go straight forward.

When the player chooses the blue cross-section, a wall blocks the character from proceeding, but choosing a cross-section that does not contain the wall will enable the player to move the character further.

The above paragraph not only highlights the premise of the game, but also its educational potential; the concept of higher dimensions is very hard to imagine and describe, and currently we're dealing only with the second and third dimensions, which are very common in the world we know. The problem begins when we try to imagine the fourth and further dimensions. Starting from the basics by playing a game should not only prove to be a fun way to learn this concept, but also a highly effective one.

The concept of a platform game is widely well known, with the first example being *Super Mario Bros*, which got numerous sequels and is widely well known. The game has some properties that should also characterize my game:

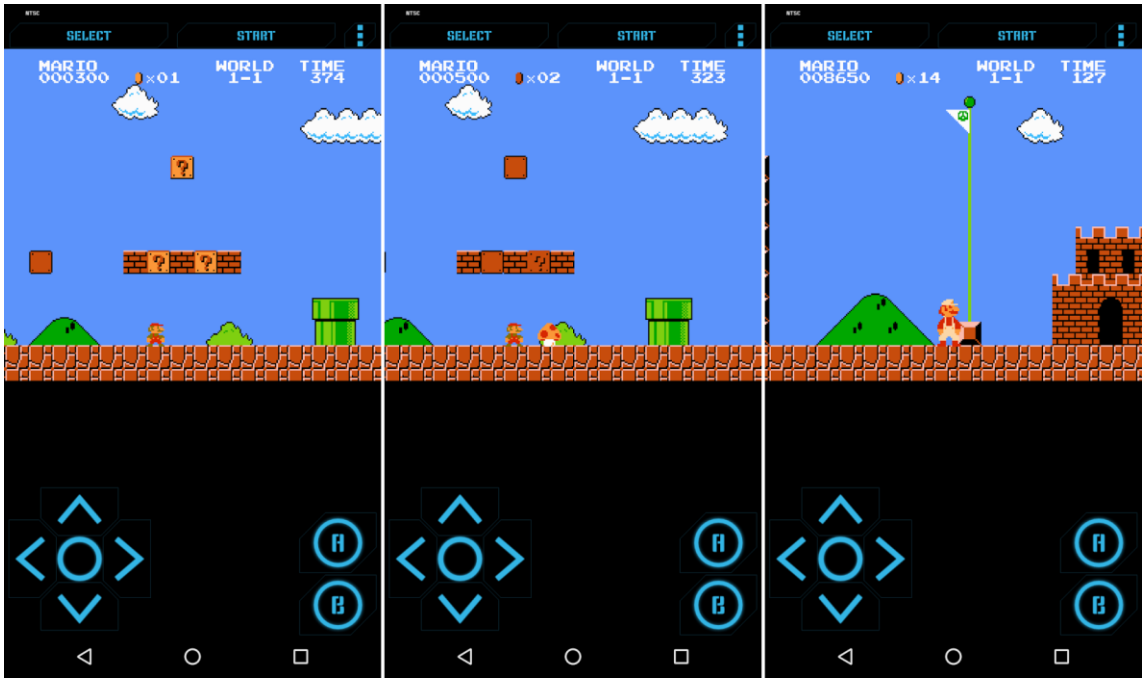


Figure 2: The original Super Mario Bros game running on a NES console emulator. The first screenshots shows the iconic design of the beginning of the game, which very quickly and neatly introduces the player to the game 's mechanics, like jumping onto platforms, jumping over obstacles, avoiding enemies and collecting bonuses (screenshot 2). Super Mario Bros. also has a very clearly defined goal (screenshot 3).

similar problems and existing solutions shown

The concept of higher dimensions has been explored in at least two games that received wider reception - Fez, in which the player moves a two-dimensional character in a three-dimensional world which he can rotate by 90 degrees, changing the point of view of the character, and Miegakure, which has not yet been released and implements a similar solution, but in four dimensions. My proposition seems similar, but gives the player more freedom, allowing him to manipulate the position of the character in the third dimension without constraints, not only to rotate the world.

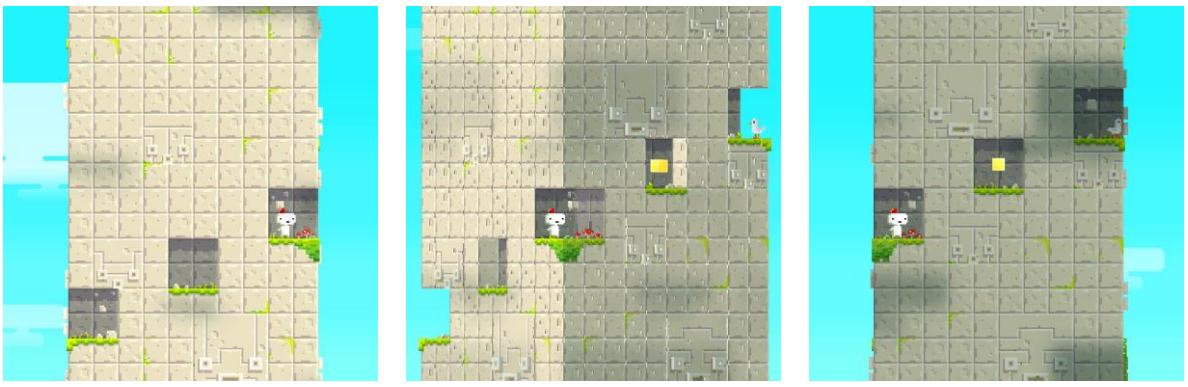


Figure 3: The 3D rotation principle in Fez[1] (taken a dimension higher by Miegakure) – the user can rotate the level, but after each rotation the level behaves just like a 2D world. This demonstrates that there are games that presented players with dimensional problems. My approach is different from them though, allowing the user to move back and forth in the higher dimension, not just to rotate the world.

The implementation of the above mechanism will require me to find or code a way of creating a 3D level map and then transcribe it into slices that could be then rendered by

[1] The screenshots come from an owned by me copy of *Fez* by Polytron Corporation.

pygame. The encoded level details will have to be stored in files that could be used in Python to create "Level" objects. And finally, the collision system used in the game should be created from scratch to account for the sudden changes in the environment when the user manipulates the third dimension.

problem shown to be solvable through a computational approach

Stakeholders

stakeholders identified

The end user could be anybody interested in puzzle games involving imagination and creative thinking, but the game can also be used for more specific needs, namely in education. The concept of higher dimensions, while interesting, is very hard to grasp. A teacher might use the game to show his students how to imagine higher dimensions, starting from a perfectly imaginable transition from 2D to 3D, and then naturally extrapolating the concept for a transition from 3D to 4D and so on. Suggesting to the students that the world in which we live could be simply a three-dimensional cross-section of a four-dimensional space could prove to be a great food for thought and an initial hook for making the students interested in advanced Mathematics. I believe that my game could be of use in the above process.

As an example of the potential end user I will use Krzysztof Oliwa, my fellow student at my college, who is keen on playing computer games as well as solving advanced Mathematics problems. This will allow me to test the game's enjoyability and helpfulness in understanding Maths concepts at the same time. I will also ask him several questions during the initial interview and I believe that his answers will lead to improvement in the game and will help me to create it in such a way that the users quickly grasp the basic concepts.

Software challenges

I decided to use Python to create the game because it is the language that I am most knowledgeable and I find it very intuitive. This will enable me to start the implementation process without the need to learn an additional language. Python is also known for its functions that allow the program to easily operate on big sets of data, which might be useful while handling the data about the levels. Python is also cross-platform and easily downloadable, making the installation of my game on pretty much any desktop computer simple. A limitation of this solution is that there might be problems with porting the game to mobile operating systems, which might not have the necessary interpreters available, and additional input methods, like a touch screen.

After choosing Python, pygame came as the natural choice for a game coding library. It is very popular, meaning that a lot of documentation and help online should be available, it is also simple, yet very flexible. It will allow me to code the game without problems, although for example a level creator would be a challenge rather hard to accomplish with it (3D rendering, modelling functions, etc.).

Creating the game consists mainly of two programming challenges. Firstly, the rendering of the cross-sections must be handled by pygame. This includes creating the levels in 3D graphics software and then exporting them in such a way that pygame can analyse the data and render it appropriately.

One of the possible solutions to this problem would be simply exporting all the data about the level mesh and making Python calculate the proper cross-section every time the user moves the character in the third dimension. While this simplifies the exporting process a lot, it also has a few obvious disadvantages. Using this approach the game will have to do a lot of complex calculations pretty much every frame, which could considerably increase processing time and decrease responsiveness of the game. In addition, it would require implementing by hand all the 3D geometric manipulations required to calculate a cross-section, while Blender (the 3D design software) has these functions already built in. It seems reasonable to use them, and thus a second solution arises - use Blender's Python scripting engine to generate and export all the necessary cross sections. The cross-sections will be stored in files that can be easily read by Python and the game could simply choose the proper cross-section from the prepared ones.

Secondly, the character's collisions with the ever-changing surroundings will have to be resolved. Pygame does have a simple collision detection system, but its greatest disadvantage is the fact that both sprites must be rectangles for it to work. Since the cross-sections will be sets of vertices, the presence of non-rectangular polygons is inevitable. In addition, the third dimension manipulation can produce some truly stunning visual effects which couldn't be achieved with rectangles only - making the design be rectangle-only would be a huge constraint on the creativity of the levels and the in-level puzzles. Therefore, a collision system that works with polygons needs to be implemented.

I did think about using a downloadable pygame library for convex shape collisions but after consideration I concluded that I will have to develop one myself due to the rapid changes the world in my game will undergo. Making an external library consider this effect could possibly be very hard and creating my own implementation will allow me to have full control over how the game works. After looking through many possible polygon collisions algorithms I have found the Separating Axis Theorem, which is considerably more understandable and easier to implement than the other options and has some features that enable the algorithm to be quite time efficient. I will describe the details of how it works in The Separating Axis Theorem section

The interview

Hi, I have come up with an idea for a game that I think you will find interesting, and I'd like to discuss it with you to get some ideas and consult my own ones. The idea is that you control a character like in a normal platform game, but there is an additional element – the level is actually three-dimensional and what you see on the screen is just a cross-section of the level.

And you can control which cross-section is currently being displayed?

Yes. I chose you for consulting since you're interested both in gaming and Mathematics, and these are the two areas that I think the game will find its use in – entertainment and introducing higher dimensions to students. Are you in?

Sure, that sound really interesting. So, what do you want from me?

I'd like you to elaborate on some questions and then do some testing when the final prototype is finished.

Ok. What are the questions then?

First, the basics. I think that we could start by discussing the controls. We need them to be slightly more complicated than normally since there is an additional dimension to handle.

That's true. I think that we could use the mouse for that, since we probably don't want the keyboard controls to be overcomplicated. By moving the mouse up and down smooth movement in the third dimension can be achieved, and then the classic navigation can be under the WSAD keys – this way it will feel natural for the experienced gamers, it's a really popular setup. You could also add arrow keys for people who just can't comply with the normal navigation. And for the WSAD, in this case the A and D keys should be for moving left and right and W for jumping. You obviously plan to add gravity?

Yes, of course. As an afterthought, do you think that requiring the use of a mouse would limit the potential audience?

Well, most of the modern games require the use of a mouse, so that should not be a problem. Either way, you can always use a touchpad if you don't have a mouse handy.

Great. Now for the looks. I was considering two approaches to the design: more natural, Super Mario like, brown soil, blue sky, clouds in the background, that sort of thing, or a modern one, utilising more toned, bluish colours.

I think that the premise of the game calls for something modern. Blue ground, orange (rust like even) character, toned, pastel colours – I think that's what you should go for.

And for the obstacles and collectibles?

Obstacles should be red, that instantly signifies danger, and collectibles should probably be yellow, even golden. Both colours will I think be easily understandable for the player.

Since we are already talking about the colours, I was thinking about a simple mechanism involving them: what if the colours changed as the player moves in the third dimension?

That's actually a great idea! It should make it easy to orient yourself in the dimension which you obviously can't see and provide quick information of your position. I think you should move from darker to lighter colours as the player moves the mouse. It will look cool and be helpful at the same time.

Do you think that a more complicated shape for the game's character should be introduced?

Your concept seems to strongly relay on geometry, so a simple square will be quite enough.

The next question is also about shapes – do you think that constructing the levels out of rectangles will be good enough or I should go for polygons? Rectangles would be easier to implement, but polygons may look nicer and help create more challenging puzzles.

In my opinion since you will already have the third dimension manipulation mechanism, which is quite mind bending in on itself, you should push the visuals to the limit. Rectangles being the only shape possible to have will limit the level design greatly and just look less cool than it could. Animated cross sections are quite awesome; I think you should embrace them in your game.

Continuing with the level design, I was thinking about introducing a timer of sorts and some collectibles, what do you think?

In my opinion what you should be aiming for is not a game concentrated on beating the levels fast, you should probably go for a more of a puzzle-centric experience. The timer could distract the player from having fun by trying to solve the level in creative ways using the mechanisms provided. I think you should concentrate on the collectibles, introduce a star collecting mechanism. It's classy and fits your needs perfectly. You place stars in various places throughout the level, make them collectible (obviously only once), show the current score at the top of the screen and make collecting all the stars the main objective of the game. This will call for creative thinking, and I believe that's exactly what you want from the players.

Precisely. Could we just clarify what do you think would be the best way to 'beat' a level?

They probably should not be scrollable (that would introduce too much motion, with the third dimension and whatnot it could become quite confusing), so it the level is the size of one screen, getting to the opposite side (in platformers it would traditionally be the right one) should be the main objective. When the player crosses the right boundary of the screen a score is displayed, along with the high score (you know, to show progress, or, well, lack of it).

And unlocking the next level should also be a reward for completing one, I presume?

Yes, that would seem natural.

How many levels do you think there should be for a start?

I don't know… I think eight is a good compromise for the beginning. It seems to be not too few and should be a good starting point. This is of course totally arbitrary.

I agree that eight is a good number for the start. I think some obstacles should also be introduced, we've talked about their colour before, but what do you think they should do?

Since the levels will be quite small introducing checkpoints doesn't make much sense. I'd say that you should just completely restart the level after colliding with lava, or however you call it.

Lava seems fitting in this case. Do you think that there should be a possibility of falling out of a level?

You could just put some lava at the bottom of the pit, it will be more consistent with the rest of the game.

Ok, before we move on to the user interface, do you have any further comments on the gameplay?

Actually yes, I have one thing. You should probably limit the speed at which the geometry of the level can change. You know, normal games sometimes have quite serious glitches with static geometry, and if you give the user direct control it could quite easily turn into quite a havoc. We definitely don't want the character to randomly go through walls or randomly land on the other side of the level, and things like that could happen if the geometry changes too quickly and the collision system freaks out. You probably want the transitions to be smoothed out.

Thanks, I'll surely take that into account. For the user experience, I'd like to discuss introducing the concept of the game to the player. It was quite easy to explain it to you using some gesticulation and your prior knowledge, but we can't just hope that the player knows something about the higher dimension concepts in advance. Therefore, I was thinking about designing the first level in a way that will introduce the mechanics smoothly.

Yeah, you probably want the first level to be easy, but containing all the gameplay elements, like lava, stars, dimensional manipulation. This way it will introduce them naturally. I still think that there should be some kind of a tutorial though. Preferably animated, it's easiest to understand higher dimensions if you show them animated.

Ok. Now the main screen. It will be the first thing the user sees and should probably also serve as a way of choosing levels.

Yes, it would be natural to let the user choose the level right at the start of the game. You could just create a set of tiles that can be clicked with the mouse. This way you should also be easily able to display the level info, like which ones are locked, the star scores, that sort of thing. You probably also want a logo there, though not really complicated – I'd imagine that your priority would be to actually finish the game.

Do you think that there should be a way of returning there at any point in the game?

Definitely! If you get stuck while trying to complete a level, you shouldn't have to close the game and open it again to try another one. I think that a simple esc key press should do the job.

Well, that's all I wanted to ask you now. Thanks for the input, it will be really helpful, and I will keep you updated on the progress!

success criteria clearly identified

Brilliant, and also thanks for the talk.

Requirement specification (success criteria)

After the interview, I listed the requirements indicated in the conversation, with **success criteria** marked in bold – those requirements can be considered critical to calling the project a success.

1. Design requirements
   - **1.1. The game's levels are created as 3D meshes and presented to the player as 2D cross-section.**
   - **1.2. Stars placed in various areas of the level to be collected by the player.**

1.3. Player's sprite is a square.
1.4. A modern-looking colour scheme that also plays on the classic colour associations: orange player, blue obstacles, yellow stars, red lava surfaces.
1.5. Winning the level requires the player to get to the right-hand side of the screen.
1.6. Touching a lava surface results in player's death – return to the level beginning and losing the stars.
1.7. Lava surfaces used to prevent the player from falling out of the level.
1.8. Levels can be chosen from a starting screen.
1.9. Player unlocks access to a level by beating the level before it.
1.10. There is a tutorial at the start of the first level chosen by the player to introduce the player to the basics of input.
1.11. The first level is designed so that it naturally helps the player understand the game's concepts.
1.12. There are eight levels in total.

2. Input requirements
2.1. WSAD keys or arrow keys used to control the player.
2.2. Mouse used to control movement in the third dimension.
2.3. Mouse used to choose a level from the main screen.
2.4. By pressing the esc key player returns to the main screen.

3. Output requirements
3.1. Main game screen shows which levels are unlocked and high scores for each level.
3.2. A star score is shown throughout the game play and stars appear empty when collected.
3.3. After finishing the level, a screen is shown displaying the high score and the current score.
3.4. A cross-section of the three-dimensional level is rendered.
3.5. Colours of the game elements change from lighter to darker with the changes in the third-dimension to provide simple information on position.

4. Processing requirements
4.1. Player should not be able to go through obstacles.
4.2. Changes in level shape should not result in random behaviour of the player's sprite.
4.3. Colliding with a star should add one to the star score.
4.4. Stars that were once collected should not be collectible again.
4.5. Colliding with a lava surface results in resetting the level.
4.6. The data about the level is imported from text files and properly processed.
4.7. Mechanisms are introduced to make the movement in the third dimension smoother.
4.8. Crossing the right edge of the screen should result in winning the level, saving a high score and unlocking the next level,
4.9. The player's sprite should fall under simulated gravity.

Additionally, based on my knowledge and research, I outline the hardware and software requirements below, with reasons for having them.

5. Hardware requirements
   5.1. Keyboard (used for controlling the movement of the player's sprite)
   5.2. Mouse (used for moving in the third dimension)
   5.3. Screen (bigger than 500x500px, the size I decided to use for the game)
   5.4. A standard computer (memory and processing requirements of Python and pygame are rather low, meaning that they should run without problems even on low-performance computers like the Raspberry Pi. After finishing the game, I will test it to establish more precise requirements in this area).
   5.5. Around 100MB of space for the Python interpreter, the pygame library and the game files.

full justification of hardware and software used

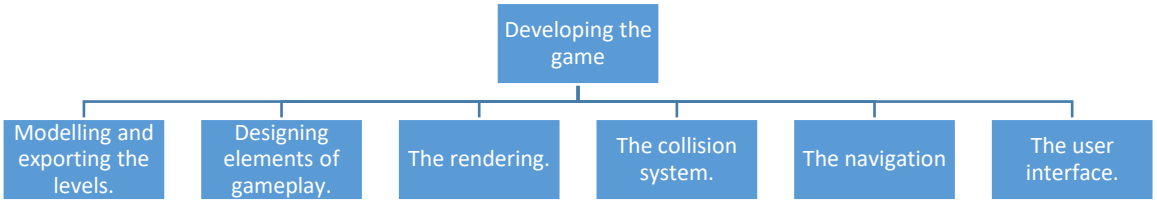
6. Software requirements
   6.1. A Python 3.0 or above interpreter (for running the game).
   6.2. The pygame library (for rendering the graphical elements).
   6.3. Any operating system supporting Python, pygame, and a GUI (Graphical User Interface), for example Windows, macOS or a Linux distribution.

Analysis 10/10

A comprehensive analysis stage that looks at the problem from a computational approach, identifies and speaks to stakeholders and looks at previous approaches.
The success criteria are clearly shown as well as the software and hardware requirements and justification.

Design

The problem (creating the game) can be split into six basic parts:



This way of decomposition will allow me to concentrate on certain parts of the game and therefore streamline my workflow – it should stop me from attempting to do everything at once. It also provides a clear framework for designing the game, which will help in the Design section.

Both in Design and in Implementation I will attempt to solve these problems separately, so that in the end they create the complete solution. Each of those parts is also divisible into smaller problems, with each having a separate section in the document.

The sections that directly concern particular requirements and success criteria have the indexes of those criteria in brackets after the title of the section (based on the section Requirement specification (success criteria)).

approach shown and problem broken down

Creating the levels

Challenges of designing the 3D model (1.1)

The game model asks for a very specific kind of level design. When creating the 3D model of a level I must keep in that the user will be in fact navigating a cross section of it, not the actual model I see. This asks for several features that should characterize the finished model:

- Cross sections should only contain convex shapes (this is due to the limitations of the collision detection system and will be discussed later).
- There should not be places where the player can get stuck, without a feasible way out.
- The level should be navigable when playing, i.e. the jumps cannot be too long or too high.
- The levels cannot be too abstract since the user has to create kind of a mental map of them to be able to pass the level.

While the user plays the game in 2D and should mentally map the level into a 3D space, I see the level in 3D while creating it, but I must consider the fact that the final, playable version, will be 2D only.

Fortunately, Blender (the 3D design software I will be using) provides its user with some features that will make the process easier, like live previews of the cross sections.

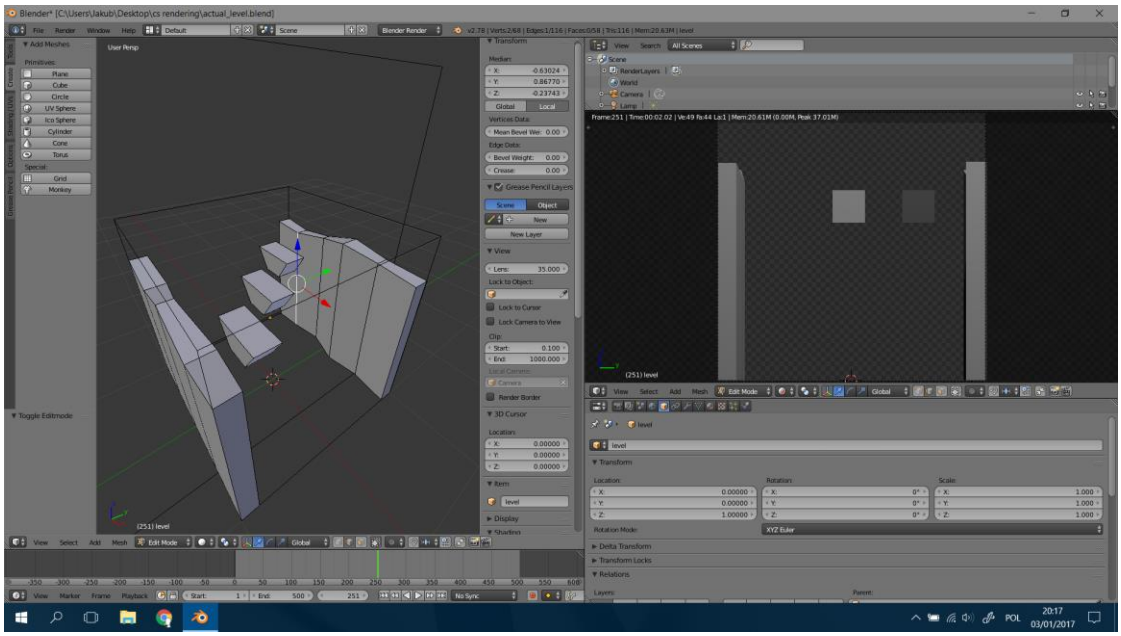


Figure 4: The Blender interface allows me to create the level while seeing it in 3D (left window), and look at a preview of the cross section at the same time (right window).

The software used

As mentioned before, I used Blender as the modelling software. I chose it mostly because of the experience that I already had with using it, and 3D modelling programmes tend to have a steep learning curve, so the effort of choosing software that is more popular in the 3D graphics industry will be quite big, with no instantly visible advantages. Blender is also well known for its open source nature and supporting community, which I considered possibly helpful during creating a non-standard project like this.

Apart from that, possibly the biggest advantage of Blender is its extensive scripting engine that lets me perform complicated operations on a 3D model using Python, a programming language that I am familiar with. The scripting engine is perfect for my use case – I need a slicing operation that outputs the model data into a text file and can be used on many different models quickly, which calls for an algorithmic solution.

Exporting the 3D model (1.1, 4.6)

When I have created the model in 3D, I need to export it in a format that Python can handle to render the level in game. The output file should be divided into segments containing data about each cross-section, and in each segment, there should be data about the polygons that the cross section contains – the x and y coordinates of their vertices, to be precise.

```
x_1 x_2
x_2 y_2
x_3 y_3

x_1 y_1
x_2 y_2
x_3 y_3
x_4 y_4

#
x_1 x_2
x_2 y_2
```

```
x_3 y_3
...
```

Code snippet 1: The structure of the level data file. The blank line signifies the end of the current polygon, and the ''#'' symbol signifies the end of the current cross section.

To obtain a file like this I will use a modifier, which is a blender function that modifies a mesh. The one I will use is called Boolean, which takes a mesh and cuts out of it a shape defined by another mesh. On the below picture it can be seen how the modifier can be used to create the cross section effect.

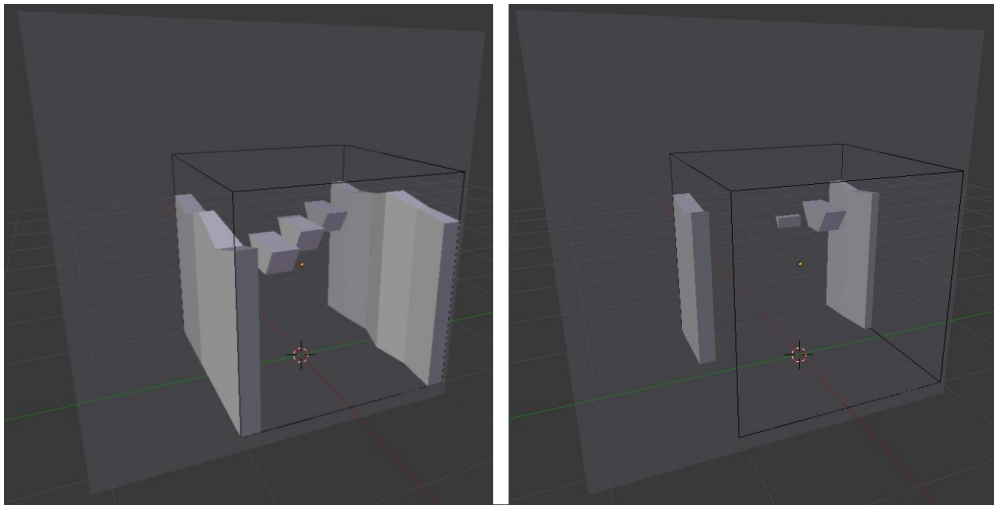


Figure 5: The level mesh with and without the Boolean modifier. The grey square is used to define the cut.

Taking the above into account the exporting process should consist of moving the cutting plane to a desired position, making a copy of the level mesh to work on, apply the Boolean modifier to it and read its data to get the desired polygons. As can be seen on Figure 5, the Boolean modifier creates a mesh that contains not only the cross section, but also additional faces that were on one side of the cutting plane. The algorithm needs to checks if all of the vertices of each face of the new mesh are on the cutting plane and only then include it in the exported data.

This can be represented as a flow chart:

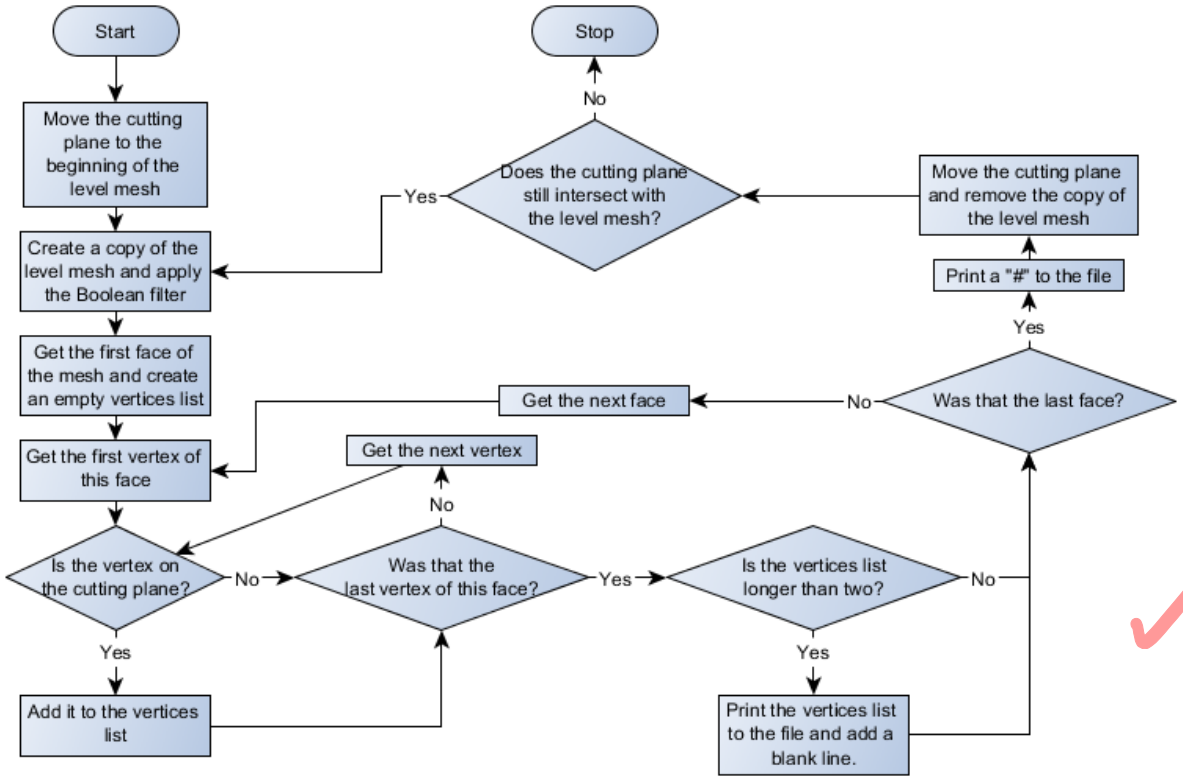


Figure 6: The level export in Blender.

And in pseudocode:

```
1    FOR i=0 TO 500:
2         cutting_plane.x = i
3         level_copy = level.duplicate()
4         level_copy.addModifier("boolean")
5         FOR face IN level_copy:
6              vertices = []
7              FOR vertex IN face:
8                   IF vertex.x == cutting_plane.x THEN
9                   vertices.append(vertex)
10             NEXT vertex
11             IF vertices.length > 2 THEN
12                  file.write(vertices + "\n")
13        NEXT face
14        file.write("#\n")
15   NEXT i
```

Code 2: The level data export from Blender in pseudocode.

The procedure creates a file with all the level data necessary to render it, which I will discuss later.

Winning conditions (1.5, 3.2, 3.3, 4.3, 4.8)

As discussed in the interview with the stakeholder, the game should adopt a classical mechanism of 'beating' the level – the user is supposed to get to the right-hand side of the screen. This can be easily detected by checking whether the player's x coordinate is bigger than the screen length.

Another important part of 'beating' a level in a game is evaluating how well did the user do. As discussed in the interview, introducing a timer contradicts the idea of the game challenging the user's mind and pushing him to think up creative solutions to the presented problems, taking his time for doing that. Instead, a star collecting system will be introduced.

The idea of star collecting is that the user is not only supposed to pass the level, but also collect objects while passing it. When the level is completed, the user is presented with how many stars he collected (usually out of three) and the high score is saved on the hard drive of the computer.

This system gratifies the creative thinking needed to get off the beaten path and collect the stars, but also does not push the user to try to complete the level as fast as possible.

A collision between the player and the star can be found by comparing their x and y coordinates, similarly as with normal obstacles (explained in The Separating Axis Theorem section), but only on two axes (we assume the star to have a square bounding box, which will increase the performance without compromising intuitiveness).

A visual indication of the number of stars collected will be given while playing the game, where a filled star means that the star has been collected and an empty star means that the star is still to be collected.

Figure 7: A rough sketch of the indication of the collected stars, with the uncollected state on the left, and the collected state on the right.

The star scoring system needs two algorithms, one that count the score and one that saves the high scores and displays them to the user.

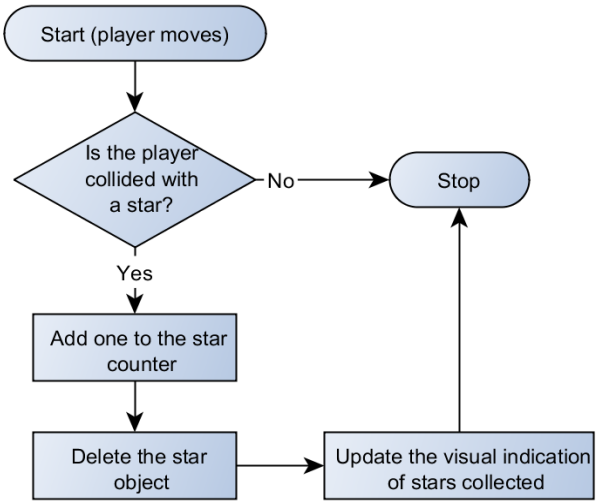


Figure 8: Star score update algorithm. It removes the star so that it cannot be collected twice and updates the score. We assume that at the start of the level the star counter is set to zero.

```
1    IF star.collided(player) THEN
2         star_counter += 1
3         star.remove()
4         updateStarNumber(star_counter)
5    ENDIF
```

Code 3: This if condition updates the star score if the player collides with a star. It is run every time the game refreshes.

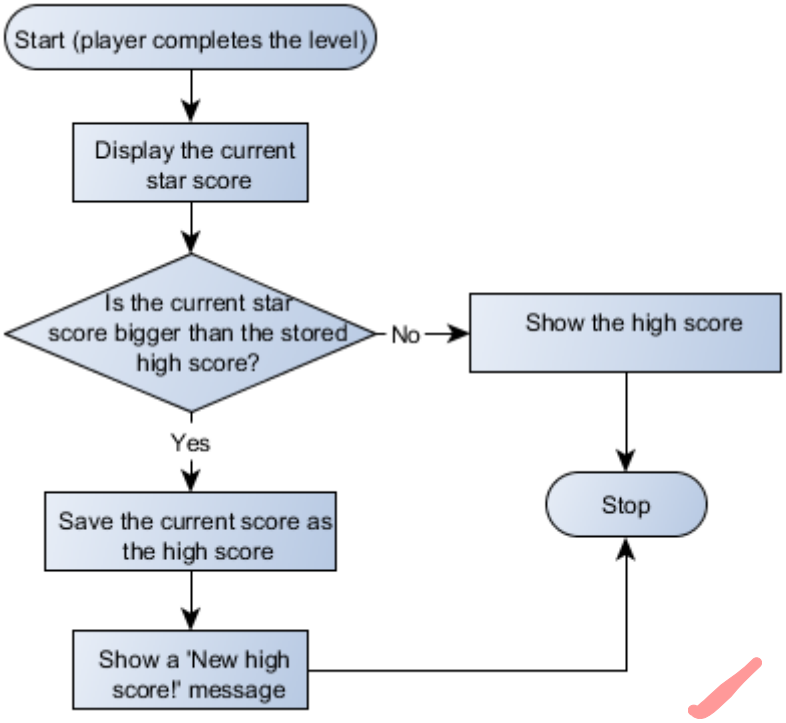


Figure 9: An algorithm that handles the updating of the high score and displaying it to the user.

```
1    showScore(star_counter)
2    IF (star_counter > star_high_score) THEN
3          saveHighScore(level_number, star_counter)
4          showMessage("New high score!")
5    ELSE
6          showMessage("The high score is: " + star_high_score)
7    ENDIF
```

Code 4: The high score updating algorithm in pseudocode.

It is not necessary to validate the star score – it will always be smaller or equal than 3 since the player does not have access to the variable itself. It can only be increased by colliding with stars and there are only three of them (colliding again with the same star is impossible since they are immediately deleted after collision).

Rendering the star score (3.2)

Displaying the star score to the user during the gameplay, as well as after the level is finished, is an important part of communication with the user. A mechanism will be implemented that displays a given number of stars (based on an integer) out of three at given x and y coordinates. This can be done in a for loop with an iterator *i* – if *i* is greater than the number of stars minus one (since the iterator starts at zero), render an empty star, if it is smaller or equal render a full star. The coordinates at which the stars should be rendered are easy to calculate by adding the star's width to the initial x coordinate and leaving y the same.

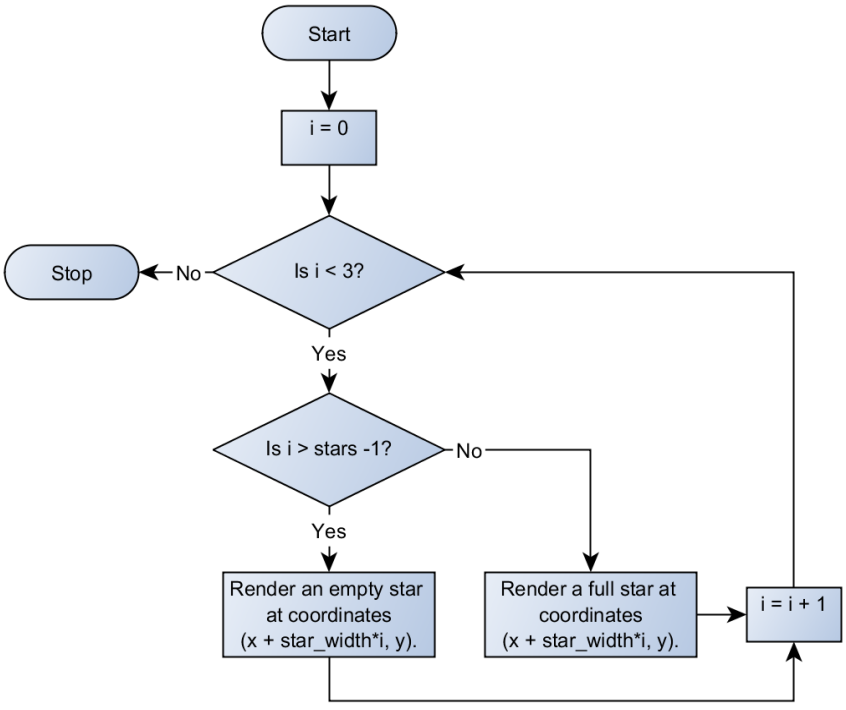


Figure 10: The star rendering algorithm. x and y are coordinates at which the rendering should be started, star_width is the width of the star bitmap.

```
PROCEDURE renderStars(stars, x, y, bitmap_full, bitmap_empty):
    FOR i=0 TO 2 :
        IF (i > stars-1) THEN
            renderBitmap(bitmap_empty, x + bitmap_empty.width*i, y)
        ELSE
            renderBitmap(bitmap_full, x + bitmap_full.width*i, y)
        ENDIF
    NEXT i
ENDPROCEDURE
```

Code 5: The pseudocode for rendering a given number of full stars at given x and y coordinates. bitmap_full and bitmap_empty are two bitmaps containing images of a full and an empty star.

To make sure that the algorithm works I traced it in a trace table (assuming that a star's width is 50px):

i	stars	stars - 1	rendering
0	2	1	Render full at (x, y)
1	2	1	Render full at (x + 50, y)
2	2	1	Render empty at (x+100, y)

The algorithm renders the correct number of full stars. And for another number of stars:

i	stars	stars - 1	rendering
0	3	2	Render full at (x, y)
1	3	2	Render full at (x + 50, y)
2	3	2	Render full at (x+100, y)

Saving high scores (4.8)

Once the player finishes the level his score should be saved and another level should be unlocked. Additionally, the state of the game should be saved on the hard drive so that the high scores and progress will survive after the user closes the game.

Each level can be in one of four states: locked, 0 stars, 1 star, 2 stars, 3 stars. The number assigned to locked can be -1, for easy checking whether the level is locked or unlocked. These numbers can be stored in a text file (separated by spaces) which will be read when level information needs to be displayed, and written to when the user completes a level and possibly establishes a new score.

```
3 1 2 0 -1 -1 -1 -1
```

Code 6: The proposed save file layout. The player collected 3, 1 and 2 stars in levels 1, 2 and 3 respectively, has not completed level 4, and thus levels 5, 6, 7 and 8 are locked.

This solution is optimal for Python since it has simple functions that can divide a string into an array using a delimiter (space in this case), and to create a string out of elements of an array, joining them by a delimiter (again, space in my case).

Obstacles (1.6, 4.5)

Apart from the level itself, which by definition of a platform game is an obstacle that the user has to pass, the game should include additional obstacles that will make the process harder. As discussed in the interview, a surface that resets the player's progress to the level's beginning will be introduced.

This will make the user consider his decisions and should also introduce a more arcade character to the game, making it progressively harder and more challenging for the user's creativity.

As discussed in the interview, these surfaces should be coloured red, which is a colour often associated with danger. In the design process, I should reference them as 'lava surfaces'.

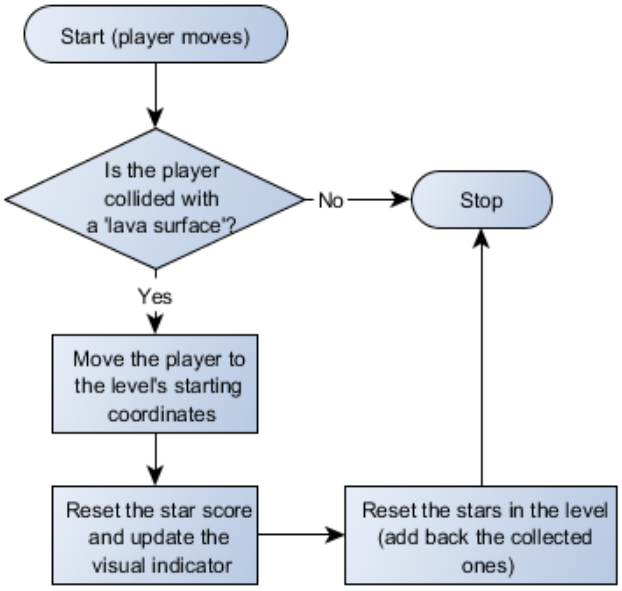


Figure 11: The algorithm that makes the player 'die' when he hits the lava surface.

```
1    IF lava.collided(player) THEN
2          player.move(x_start, y_start)
3          star_counter = 0
4          stars.reset()
5    ENDIF
```

Code 7: The pseudocode that checks if player 'died' in this game refresh and resets the level if he did.

Colour scheme (1.4, 3.5)

As discussed in the interview, the colour scheme of the levels should be clean and futuristic, making the walls and the background blue. In contrast to that, the player should be coloured orange, similarly to rust. The stars will be yellow, as they are mostly presented like that in other games and that will be a simple visual indication of a collectible for the player. As discussed above, the lava surfaces will be red, which is a colour associated with danger.

One feature discussed in the interview is the changing of colours with the change in the third dimension – as the user changes the visible cross section, the colours of the level also

change, giving the user a visual indication of his position in the third dimension for simple orientation.

For this mechanism to work I needed a smooth colour palette with nice progressions and a way to easily calculate which colour to use based on the current position in the third dimensions. For the colour palette I chose Google's Material Design Colour Palette[2], which provides a versatile library of pleasant to the eye pastel colours that look nicely together. Then I made created gradients for each important element that will appear in the level:

Colour scheme for the level

Colour scheme for the background

Colour scheme for the player

Colour scheme for the stars

Colour scheme for the lava surfaces

Calculating which colour to choose is rather simple since colours are represented in Python as tuples of form (red, green, blue), where the values for red, green and blue range from 0 to 255. For every element, we have two boundary colours, for example for the level they are (33,150,243) and (13,71,161). In the calculations, we can consider each basic colour separately. For example, for the level the boundary values for red are 33 and 13. Now we just have to choose a value in between based on our position in the third dimension:

$$colour\ value = min + \frac{z}{width} \times (max - min)$$

Where min is the bottom boundary, max is the top boundary and width is the width of the level in the third dimension.

I checked the above colours using special filters and they should be distinguishable by people with the three most common types of colour blindness (deuteranomaly, protanomaly,

[2] https://material.io/guidelines/style/color.html

tritanomaly), which is an important usability feature since it would be impossible to complete the game if one is unable to distinguish between normal obstacles and lava.

Design mock-up

Taking all the above points into consideration I have drawn a simple mock-up of a game screen that I will use during the implementation process for reference.

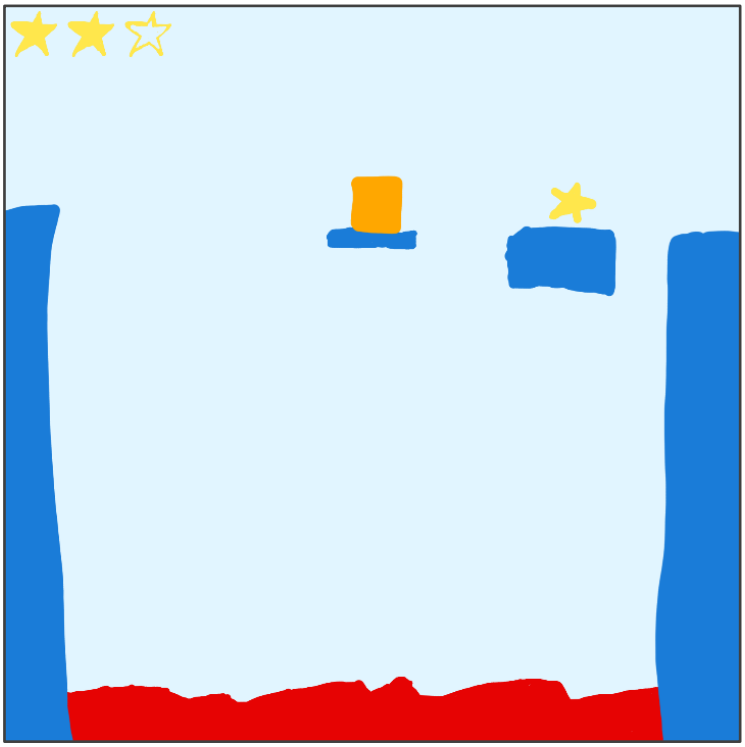


Figure 12: The first mock-up of the game.

Rendering the level

Once the level is designed another challenge is bringing it to life in the game.

Importing data from the text files (4.6)

As discussed in Exporting the 3D model section, the Blender exporting process gives us a text file containing all the data needed to render the level in 2D. To use this data, we need to first load it into Python.

To accomplish that, I introduce a three-dimensional array data, with the structure:

$$data[zIndex][polygonIndex][vertexIndex][xory]$$

Where *zIndex* is the position in the third dimension, *polygonIndex* is the index of a polygon in the cross section, *vertexIndex* is the index of that polygon, and *xory* is the reference to each coordinate of that vertex: 0 is the x coordinate and 1 is the y coordinate.

This structure gives me simple access to the level shape's data and makes working with it easy thanks to Python's for loops that allow me to very comfortably iterate on an array elements.

To fill the array in I have to take the data file and read it line by line deciding what to do based on the line contents. The file structure is specified in the Exporting the 3D model section: new line means end of a polygon definition, '#' means end of a cross section definition.

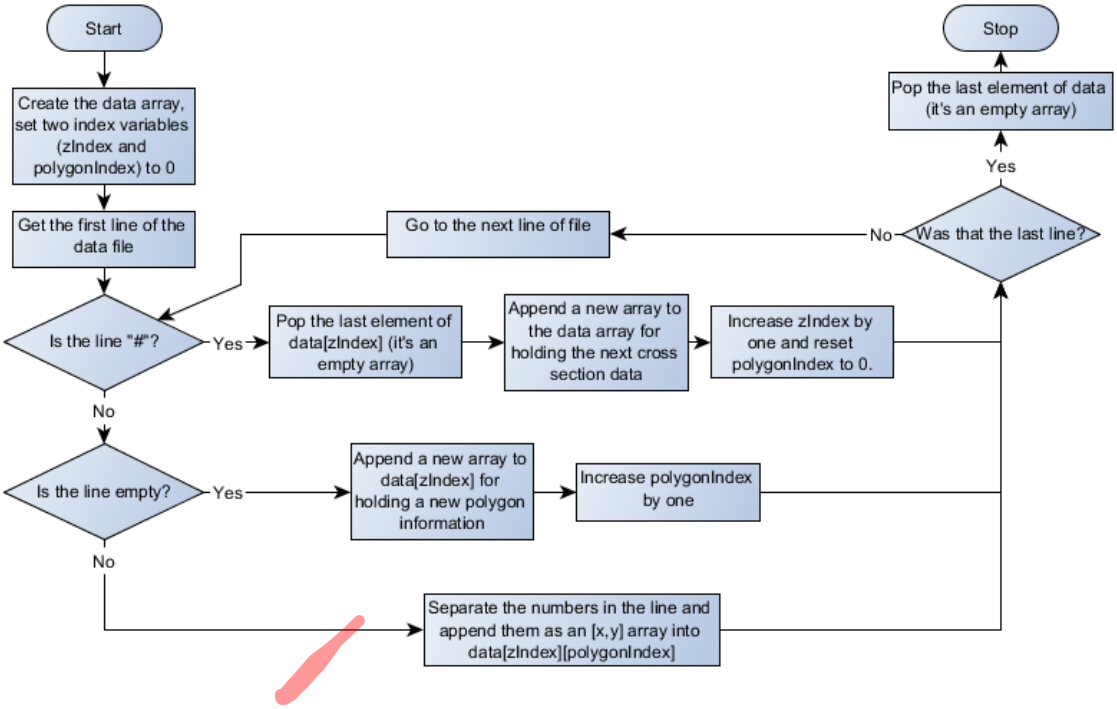


Figure 13: The importing algorithm.

```
1    data = [[[]]]
2    polygonIndex = 0
3    zIndex = 0
4    FOR line IN "file.txt" :
5         IF (line == "#\n") THEN
6              data[zIndex].pop()
7              data.append([[]])
8              zIndex += 1
9              polygonIndex = 0
10        ELSE IF line == "\n" :
11             data[zIndex].append([])
12             polygonIndex += 1
13        ELSE
14             data[zIndex][polygonIndex].append(line.split(", "))
15        ENDIF
16   NEXT line
17   data.pop()
```

Code 8: The importing algorithm written in pseudocode. The split() function creates an array out of a string dividing the string using a given delimiter.

To check whether the above algorithm works I used a trace table and a sample file:

```
146 371
151 348
154 333

#
147 371

33 334
39 308

#
```

Code 9: A sample level data file. Some of the polygons have less than three vertices for readability of the trace table.

Line of the file	zIndex	Polygon Index	data
----	0	0	[[[]]]
146 371	0	0	[[[[146,371]]]]
151 348	0	0	[[[[146,371][151,348]]]]
154 333	0	0	[[[[146,371][151,348][154,333]]]]
	0	1	[[[[146,371][151,348][154,333]][]]]
#	1	0	[[[[146,371][151,348][154,333]]][[]]]
147 371	1	0	[[[[146,371][151,348][154,333]]][[[147,371]]]]
	1	1	[[[[146,371][151,348][154,333]]][[[147,371]][]]]
33 334	1	1	[[[[146,371][151,348][154,333]]][[[147,371]][[33,334]]]]
39 308	1	1	[[[[146,371][151,348][154,333]]][[[147,371]][[33,334][39,308]]]]
	1	2	[[[[146,371][151,348][154,333]]][[[147,371]][[33,334][39,308]][]]]
#	2	0	[[[[146,371][151,348][154,333]]][[[147,371]][[33,334][39,308]]][[]]]
(EOF)	2	0	[[[[146,371][151,348][154,333]]][[[147,371]][[33,334][39,308]]]]

Table 1: A trace table demonstrating the import of a data file containing level vertices coordinates.

While the array does not look legible written in one line, it becomes clear that the algorithm works if we format the output properly:

```
[
  [
    [
      [146,371][151,348][154,333]
    ]
  ]
  [
    [
      [147,371]
    ]
    [
      [33,334][39,308]
    ]
  ]
]
```

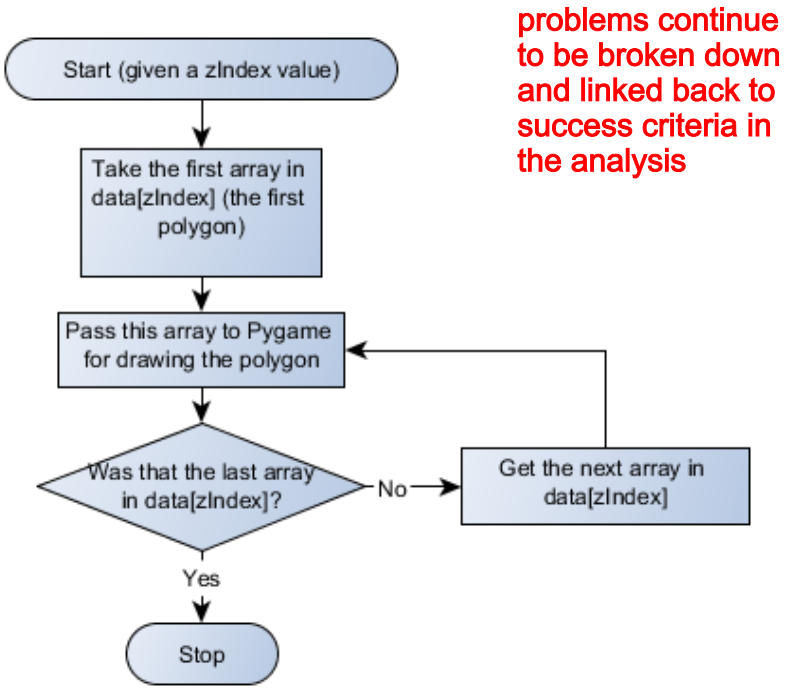
The first level is the data array itself, the second level are the arrays for different cross sections, the third level are the polygons within each cross section and the fourth level are the vertices of each polygon.

Now that I have an array with the level data I can proceed to rendering it.

Rendering a single cross section (3.4)

Before going into animation and giving the user control over which cross-section to render I should create a mechanism that renders a single cross section with a given zIndex. This should be fairly straightforward thanks to the array created in the previous section.

The data needed to render a particular cross section is contained in data[zIndex]. This contains an array for every polygon in the cross section, which contains arrays with coordinates of its vertices and pygame can use these to render a polygon. The cross section rendering algorithm will thus look like this:



problems continue to be broken down and linked back to success criteria in the analysis

Figure 14: Rendering a level given a zIndex.

```
1    FOR polygon IN data[zIndex]:
2         renderPolygon(polygon)
3    NEXT polygon
```

Code 10: The pseudocode representing an algorithm that renders a single cross section.

In the example from the Importing data from the text files section data[1] will for example return

```
[
  [
    [147,371]
  ]
  [
    [33,334][39,308]
  ]
]
```

And we can take data[1][0] (which will be [[147,371]]) and data[1][1] (which will be [[33,334][39,308]]) and pass them to pygame for drawing.

further detail given about usability features

Giving the user control (2.2)

The user will be give control over the movement in the third dimension, meaning that he can choose which cross section will be displayed. As discussed in the interview, the process of choosing the cross section should be fluid, so the natural input device for that is the mouse, which is analogue by nature, unlike the keyboard. By moving the mouse up and down the user will be able to choose the cross section.

Having the cross section rendering mechanism I could just take the y coordinate of the mouse and render the cross section given by it (assuming that the level is cubic, i.e. it's as wide in the third dimension as it is high and wide in 2D).

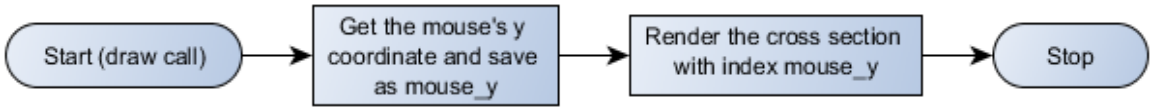


Figure 15: A simple rendering mechanism taking the input from the user.

This solution has numerous flaws. Firstly, the transition will not be fluid – the mouse can move very fats, meaning that it can move through the whole screen in a matter of a second, making the transition jumpy and seemingly random, which will not be very aesthetically pleasing. Additionally, changes in the level so sudden and random could confuse the collision engine producing very unexpected results.

To conclude, the cross section choice and animation mechanism needs to be more complex in order to be more smooth and easy on the game's engine.

Improving the controls (4.7)

A system that solves the above problem should have the following properties:

- The bigger the difference between the current state and the state desired by the user, the faster the change should be.
- The animation should ease out while reaching the desired position to avoid confusion.
- There should be a maximum change speed so that the animation always looks smooth and the game's collision engine does not get confused.

The first two features can be accomplished by calculating the difference between the current state and the state desired by the user and multiplying it by a factor smaller than one. The third feature can be accomplished by checking whether the absolute value of the result is bigger than a constant and if so, changing it to the constant.

The absolute value function (denoted as abs(x)) is important in implementing the second check. An absolute value of x is its non-negative value without regard to its sign. Since the difference calculated in the first step has a direction (up or down) which is signified by a sign, if we want to compare its magnitude to a constant we must first calculate the difference's absolute value and then compare it with the constant. We cannot forget about the direction when making the difference equal to a constant – it needs to inherit the sign of the difference.

To do that we can multiply it by $\frac{diff}{abs(diff)}$, which is always equal to 1 or -1 and has the sign of *diff*.

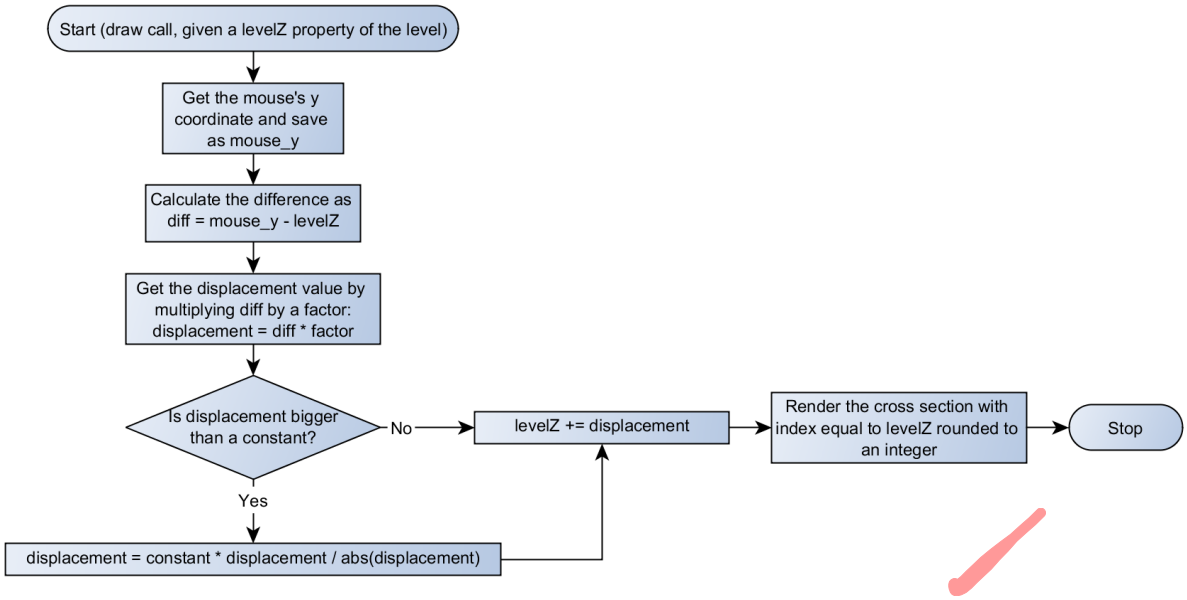


Figure 16: The new cross section choice and animation mechanism

```
1    mouse = getMousePosition()
2    diff = mouse.y - levelZ
3    displacement = diff * factor
4    IF (displacement > 5) THEN
5         displacement = 5*displacement/abs(displacement)
6    ENDIF
7    levelZ += displacement
8    renderCrossSection(floor(levelZ))
```

Code 11: The new cross section choice and animation mechanism represented in pseudocode. floor() is a function that returns the passed variable rounded down.

Note that a new variable was introduced: *levelZ*. Since the displacement will often not be an integer we cannot store the position in the third dimension as zIndex, which is used for indexing arrays in the data array. LevelZ is a float that is rounded just before the rendering operation, giving the desired zIndex and solving the problem.

The Separating Axis Theorem

The separating Axis Theorem is the principle on which the collision system used in the project will be based.

Why write a custom collision system?

The sprite collision system in pygame is only capable of detecting collisions between rectangular objects, which is a major limitation. First I considered making everything in game rectangular, but I concluded that it would defeat the purpose of having a complex third dimension manipulation system which looks best for more abstract shapes. This conclusion was confirmed during the interview.

The second option I considered was to use a pygame library like pylygon[3], a library that would allow me to create polygon objects with collisions. The problem with that approach was that my game is supposed to be very dynamic, the polygons constantly changing while the player changes the cross sections, and that could be very hard to account for using external libraries.

I finally decided to write my own collision detection system that would not only account for the constantly changing environment, but also benefit from the level data organization structure I decide to use (described in the Exporting the 3D model section).

The premise of the SAT

The main idea of the SAT is simple: if we can find an axis along which the projections of two convex shapes do not overlap, then the shapes also do not overlap.

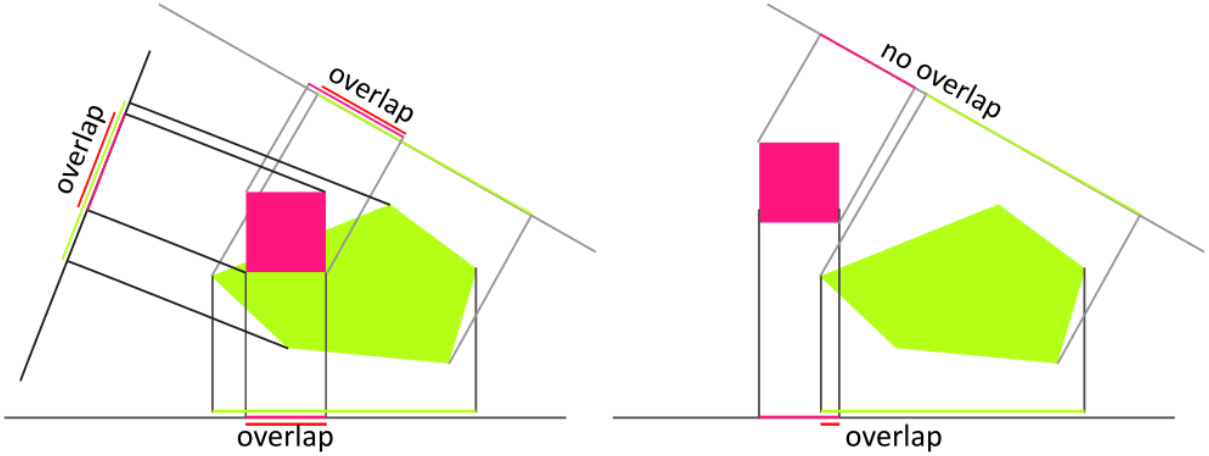


Figure 17: On the left picture, the shapes do overlap and it can be easily seen that there is no axis on which the shapes' projections would not overlap. On the right picture, the shapes do not overlap and it can be seen that there exists at least one axis on which the projections of the shapes do not overlap.

Of course, there is an infinite number of possible axes to check and that would be impossible to do programmatically. Fortunately, another insight the SAT gives us is that if there exists a separating axis (one on which the projections do not overlap), then it will be perpendicular to one of the polygon's edges.

Since the player's sprite is a square the axes we need to consider are the x axis, the y axis and the axes perpendicular to the edges of the elements of the level.

Another practical feature of the SAT is that if the algorithm sees that there is no overlap on one of the axes it can stop checking the others since only one axis without overlap is enough to conclude that the shapes do not collide. That helps with reducing the computational complexity of the SAT since the most computation-heavy case is the rarest one, when the objects actually do touch each other.

[3] http://www.pygame.org/project-pylygon-1718-3432.html

Projection vectors

If the projections of the polygons overlap on all the axes we have found a collision. The question is: what next? This is an important part of a collision detection system, even if it only has ‘detection’ in its name. We do not want jut a boolean answer to the question ‘Do the objects collide?’. We also need to do something if a collision occurs to stop it from occurring.

The simplest way to deal with a collision is to find a projection vector which will give us the distance and direction in which we have to move the player’s sprite to move it out of the collision in the shortest distance.

The SAT gives us that the collision vector is given by the axis with the smallest overlap, the direction being the direction of the axis and the distance being the size of the overlap.

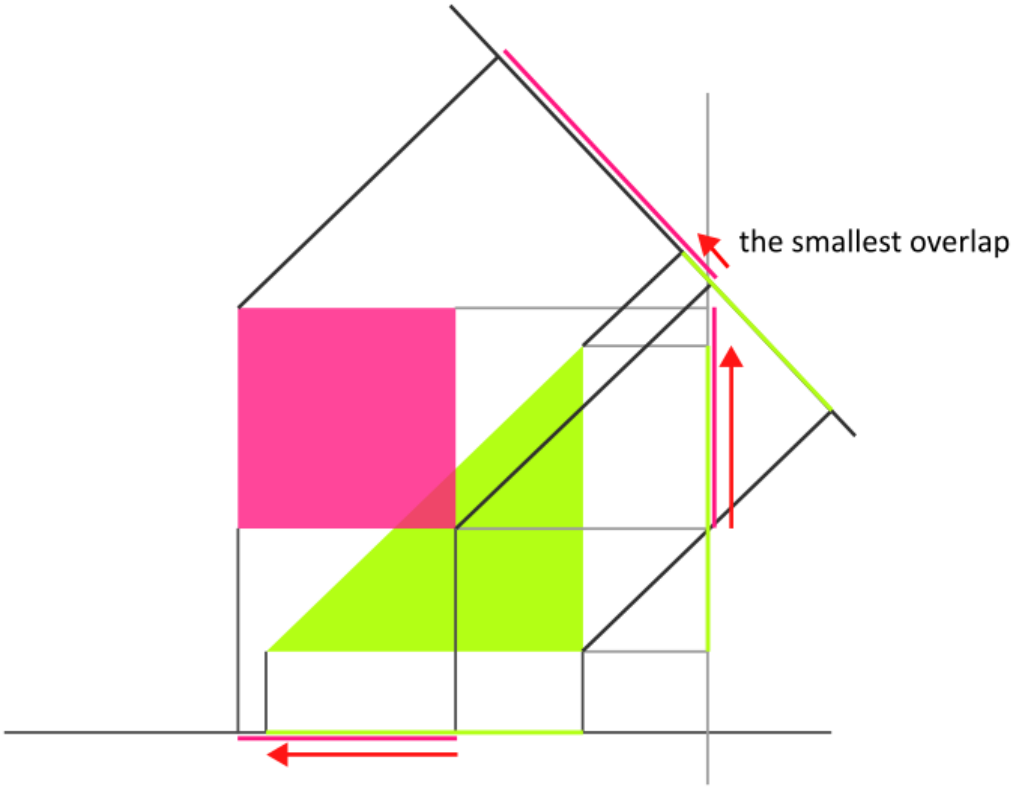


Figure 18: Visualization of the projection vectors. The smallest one will be chosen as the displacement vector. It matches the common sense, meaning that the model is suitable for physics simulations.

Introduction to vector Mathematics

The use of vector mathematics is very important in this problem and makes the calculations easier than if they were done for example by analytical geometry. I will not go to deep into the premises of vector based Mathematics in this section since the more complex concepts will be introduced as needed later in the document, this paragraph’s purpose is mainly to introduce the fact that the vectors will be used and explain how they will be defined. This can be best presented on an example picture:

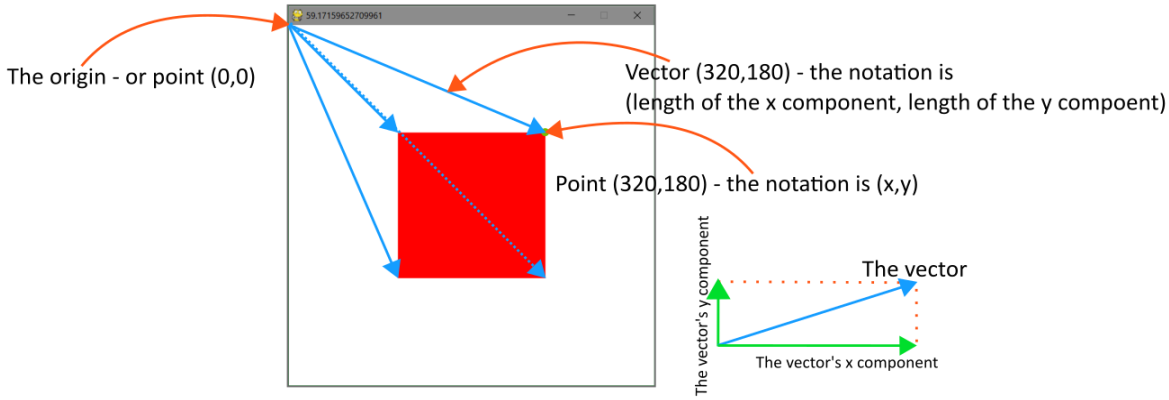


Figure 19: Short introduction to vectors and how they are defined in the game. Each vertex of a polygon can be assigned a vector from the origin to the vertex. The vector's x component and y component are equal in length to the point's x and y coordinate respectively.

Another important concept is the dot product, which can be described as a way to represent the relative orientation of two vectors. For example if the vectors are perpendicular, then their dot product is zero, and if the angle between them is zero, the dot product is maximal. It can be mathematically defined as:

$$\boldsymbol{a} = (x_a, y_a)$$
$$\boldsymbol{b} = (x_b, y_b)$$
$$\boldsymbol{a} \cdot \boldsymbol{b} = x_a x_b + y_a y_b$$

Note that vector names are written in bold in print.

The length of a vector can be calculated using Pythagorean theorem:

$$|\boldsymbol{a}| = \sqrt{x_a^2 + y_a^2}$$

Projecting polygons onto an axis

A crucial part of the SAT is the projection of a shape onto an axis. I am going to accomplish that using vector Mathematics.

We first need to have a way to define an axis. In my game, it will be the easiest to define an axis using a unit vector (of length one) starting at the origin. Its direction is the direction of the axis. Let us call this vector **n** (as in normal).

It turns out that a projection of a vector onto a line which direction is given by a unit vector is defined as:

$$x_p = (\boldsymbol{p} \cdot \boldsymbol{n}) * x_v$$
$$y_p = (\boldsymbol{p} \cdot \boldsymbol{n}) * y_v$$

Where **p** is the projected vector, **n** is the vector defining the line and **v** is the vector we are projecting. If we take the vector of each vertex of a polygon and project it onto a line the result will be as follows:

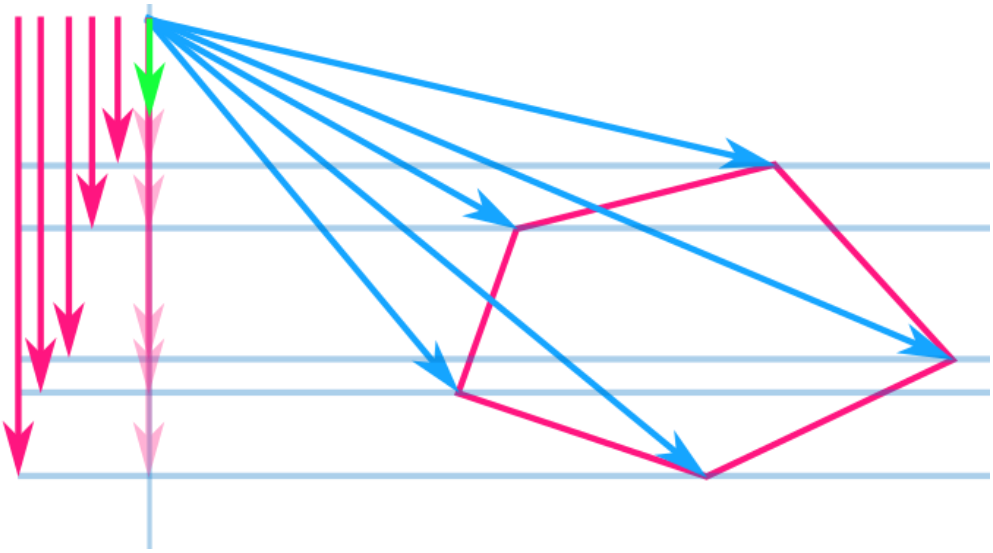


Figure 20: The projection of a polygon's vertices' vectors onto a line given by a unit vector (green). The vectors are offset for readability; they should all be on the line.

Since all the vectors are on the same line from now on we can consider only their lengths with added direction[4] (more on calculating vector lengths in the Introduction to vector Mathematics section). This thought process is similar to changing the line for a number line with a zero in the origin and checking where the end of the vector lands.

Another very useful observation is that the minimum and maximum values obtained this way are the boundaries of the shape's projection. We can thus define the projection as a section on the number line! The format for writing this in this document will be (minimum boundary, maximum boundary).

The algorithm for finding a projection of a polygon onto a line given by a unit vector is:

[4] The sign assigning is needed since the length of a vector is by definition a positive number, but the end of the projected vector might actually get into the negative side of the number line (about which line I will talk in the next sentence). To account for that I calculate the dot product of the projected vector and the line-defining vector. The dot product will be by definition negative if the vectors are facing in the opposite direction. By copying the sign of the dot product to the length of the vector I get its end's actual position on the number line.

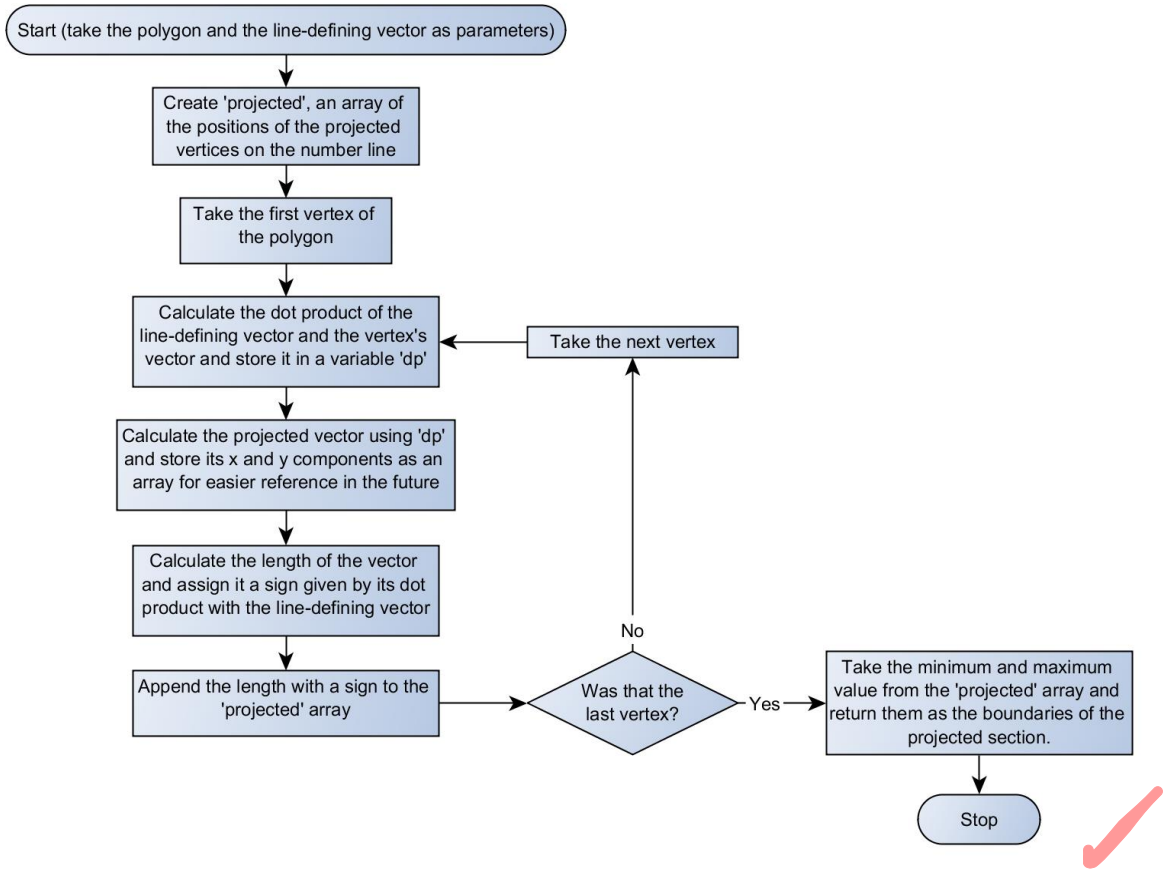


Figure 21: The polygon projecting algorithm.

```
1     FUNCTION project(polygon, normal):
2           projected = []
3           FOR vertex IN polygon:
4                 dp = vertex[0] * normal[0] + vertex[1] * normal[1]
5                 projected_vector = [normal[0] * dp, normal[1]*dp]
6                 projected_length = sqrt(projected_vector[0]^2 +
projected_vector[1]^2)
7                 // The sign is added using the method mentioned
8                 // in the Improving the controls section
9                 dp_projected_normal = projected_vector[0] * normal[0] +
projected_vector[1] * normal[1]
10                projected_length = projected_length * dp_projected_normal /
abs(dp_projected_normal)
11                projected.append(projected_length)
12          NEXT vertex
13          RETURN [min(projected), max(projected)]
14    ENDFUNCTION
```

Code 12: The pseudocode implementation of the polygon projection algorithm. The min() and max() functions return the biggest and the smallest elements in a passed arrauy

Detecting overlap

Since I now have a projection algorithm I can easily project the obstacle and the player's sprite onto the same line, getting two segments of a number line. Now I need to check whether they overlap and if so, calculate the projection vector.

There are five possible relative positions of the obstacle's and player sprite's projections:

Figure 22: Possible relative positions of the player's sprite (pink) and the obstacle (green).

As can be seen on the above Figure, the situations in which the shapes overlap are more complex than the ones with no overlap. In fact, the no-overlap situations can be easily detected by comparing the edges marked with grey circles on the Figure: if the maximum boundary of the player's sprite is smaller than the minimum boundary of the obstacle then the player is on the obstacle's left side, not touching it. Similarly, if the minimum boundary of the player's sprite is bigger than the maximum boundary of the obstacle, the player is on the right side of the obstacle and is not touching it. If neither of these conditions is satisfied there is an overlap.

Getting the projection vector

Now that we know that there is an overlap on a certain axis we need to calculate the projection vector. There are always two of them possible and one is of course better than the other:

Figure 23: There are always two possible projection vectors.

It can be seen that their length is given by the maximum boundary of the player's sprite's projection minus the minimum boundary of the obstacle's projection in one case, and by the maximum boundary of the obstacle's projection minus the minimum boundary of the player's sprite's projection in the other. The vector's value is usually written as negative if it's facing towards the negative side of the number line.

Having the lengths of the vectors we can easily compare them, but when we choose the shortest one we still need to know in which direction it is facing to move the player's sprite out of the collision. Fortunately, vectors can be multiplied by scalars – we get a vector facing in the same direction (or the opposite if the scalar was negative) and its length is multiplied by the scalar. Since the line on which we projected the shapes is defined by a unit vector, if we multiply it by the value of the projection vector we will get a vector facing in the right direction and with the correct length!

$$\boldsymbol{a} - vector; b - scalar$$
$$b * \boldsymbol{a} = (b * x_a, b * y_a)$$

The final algorithm returning the projection vectors and their lengths will look like this:

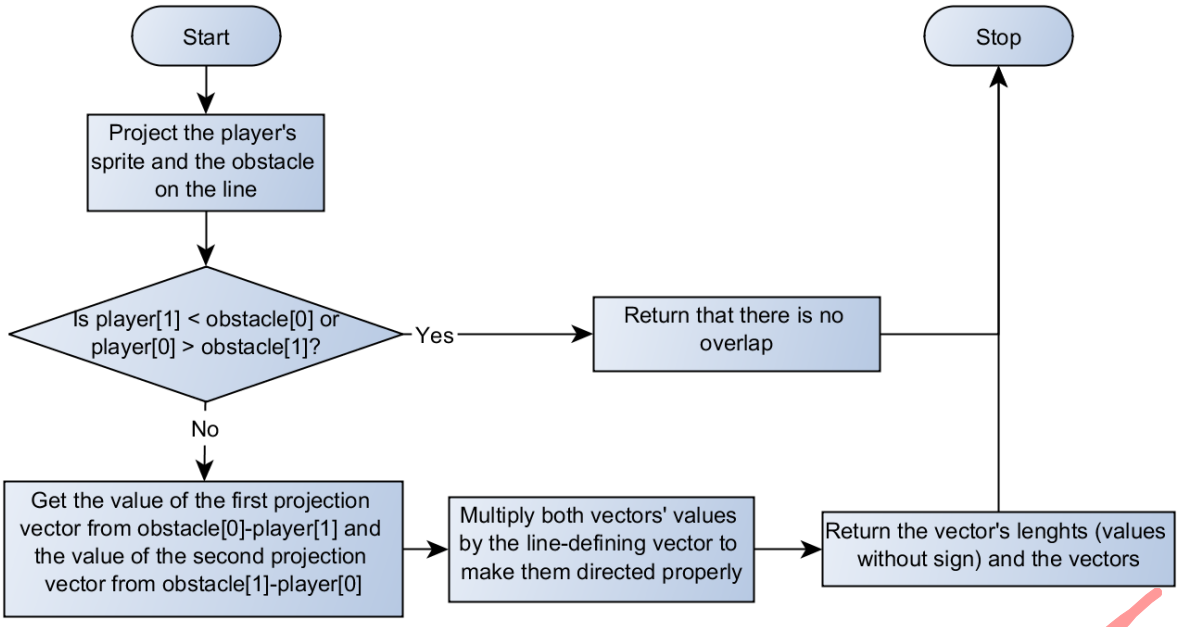


Figure 24: Obtaining the projection vectors. Note that player[0] means the minimum boundary of the player's sprite's projection and player[1] means the maximum boundary, similarly for the obstacle.

```
1    FUNCTION calculateProjectionVectors(obstacle, player, normal):
2         obstacle_projection = project(obstacle, normal)
3         player_projection  = project(player, normal)
4         IF (player[1] < obstacle[0] OR player[0] > obstacle[1]) THEN
5              RETURN FALSE
6         ELSE
7              value1 = obstacle_projection[0] - player_projection[1]
8              value2 = obstacle_projection[1] - player_projection[0]
9              vector1 = normal * value1
10             vector2 = normal * value2
11             RETURN [abs(value1), vector1, abs(value2), vector2]
12        ENDIF
13   ENDFUNCTION
```

Code 13: Pseudocode function that returns the projection vectors given a list of vertices for both the player and the obstacle. Note that I assume that an array multiplied by a constant gives an array in which every element was multiplied by that constant.

Creating the axes

As mentioned before, there is an infinite number of axes that could be checked for overlap, but the SAT states that it is enough to check the axes perpendicular to the polygon's edges. To obtain these axes I will also use vector Mathematics.

In the Introduction to vector Mathematics section I mentioned the idea that the vertices of a polygon have vectors assigned to them – these vectors start at the origin and end at the vertex. Now I will introduce the concept of vectors assigned to edges and the method of obtaining them.

First we need to discuss vector summation and subtraction. To sum two vectors we can move one of them so that it starts at the point where the second one ends and draw a new vector from the beginning of the first vector to the end of the second one:

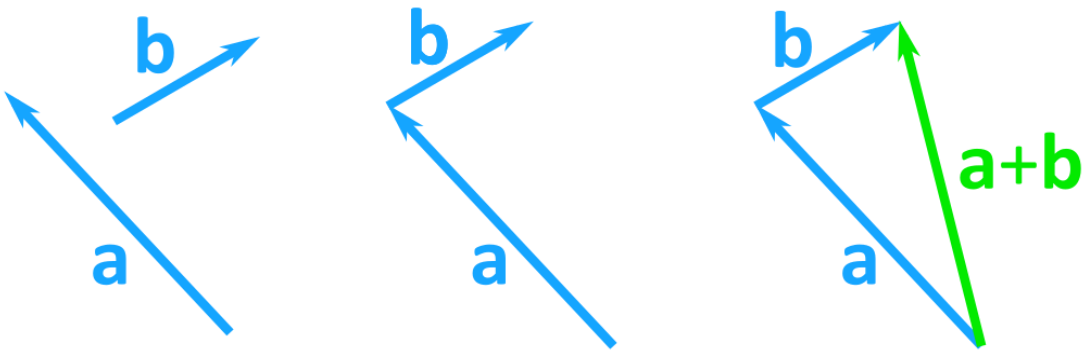


Figure 25: The principle of summing two vectors.

Algebraically this can be written as:

$$\boldsymbol{a} = (x_a, y_a), \boldsymbol{b} = (x_b, y_b)$$
$$\boldsymbol{a} + \boldsymbol{b} = (x_a + x_b, y_a + y_b)$$

Subtracting two vectors is quite similar, we just need to reverse one of the vectors since

$$\boldsymbol{a} - \boldsymbol{b} = \boldsymbol{a} + (-\boldsymbol{b})$$

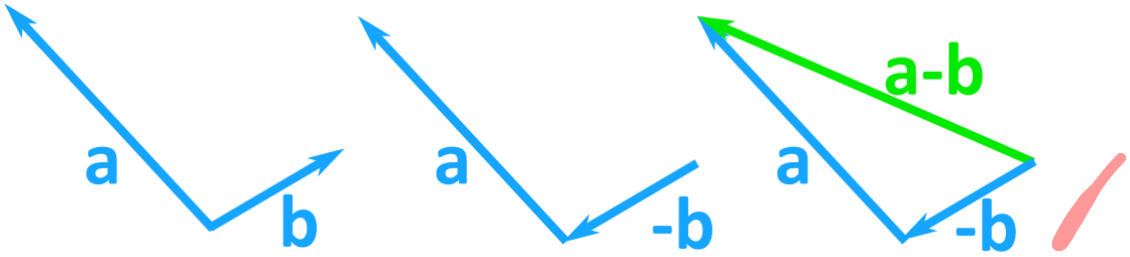


Figure 26: The principle of subtracting two vectors.

Now we can notice that the **a**-**b** vector joins the ends of the **a** and **b** vectors. Thus vector subtraction is what we need to obtain the edge vectors of a polygon.

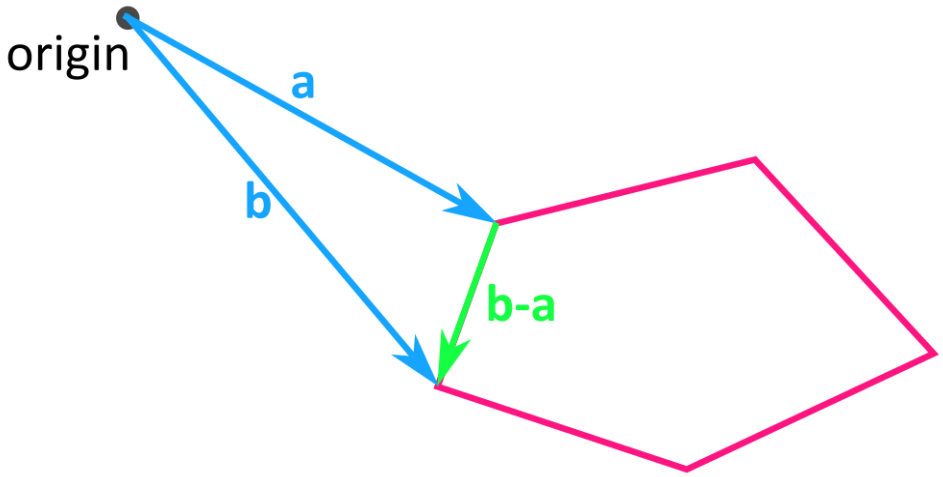


Figure 27: The edge vector

The x and y components of the edge vector can be obtained using the formula on the top of this page:

$$\boldsymbol{e} = (x_b - x_a, y_b - y_a)$$

Now we just need to turn that vector into a unit vector perpendicular to the edge and we have the axis! First, we should multiply it by the inverse of its length, reducing it to length one (vector multiplication by a scalar was covered in the Getting the projection vector section). Then we obtain the perpendicular unit vector (which is called the **normal vector**) knowing that:

$$\boldsymbol{a} = (x_a, y_a)$$
$$\boldsymbol{a}^{\perp} = (-y_a, x_a)$$

We can move this vector to the origin (the direction stays the same). Now we have the vector defining the projection axis!

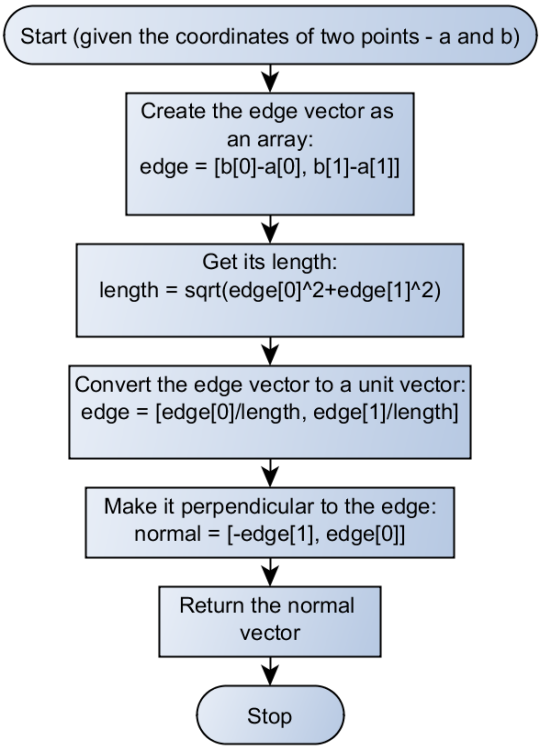


Figure 28: The procedure for getting the normal vector given two vertices forming an edge.

```
1    FUNCTION getNormal(a, b):
2         edge = [b[0] - a[0], b[1] - a[1]]
3         length = sqrt(edge[0]^2 + edge[1]^2)
4         edge = edge/length
5         normal = [-edge[1], edge[0]]
6         RETURN normal
7    ENDFUNCTION
```

Code 14: The pseudocode function that returns a vector normal to an edge defined by two points, a and b.

Putting it all together (4.1)

The procedures and functions described in the above sections are all that is needed to implement a Separating Axis Theorem based collision detection and resolving system. Now I just need to connect the pieces.

Consider a situation in which we have a player's sprite and a polygon that is the obstacle. We need to go to every edge of the polygon, calculate the normal vector, project the player's sprite and the obstacle on the line given by the normal vector, detect overlap and create the projection vectors. When we did it for all the edges we will have a list of all the possible projection vectors and their lengths. Now we choose the shortest one and translate the player's sprite by the vector so that it no longer overlaps with the obstacle.

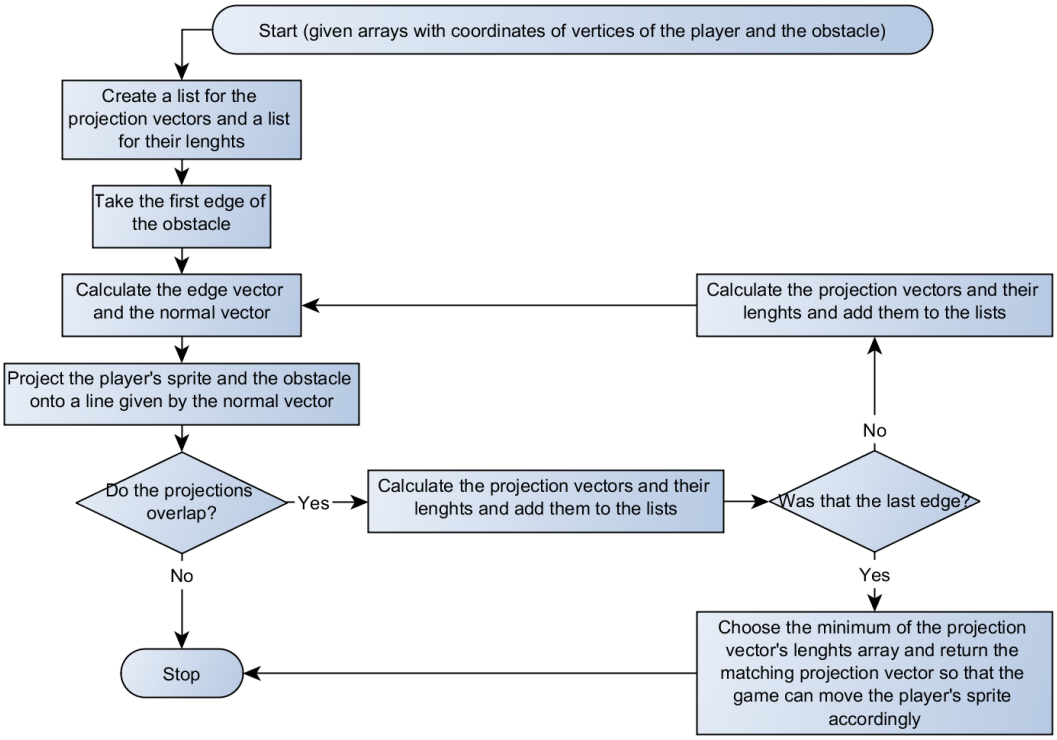


Figure 29: The SAT algorithm.

```
1    FUNCTION collide(obstacle, player):
2         projection_vector_lenghts = []
3         projection_vectors = []
4         FOR i TO len(obstacle) :
5              normal = getNormal(obstacle[i], obstacle[(i + 1) % len(obstacle)])
6              vectors = calculateProjectionVectors(obstacle, player, normal)
7              IF vectors = FALSE THEN
8                   // Thanks to the SAT we can break when there is no overlap on
at least one axis.
9                   RETURN [0,0]
10             ELSE
11                  projection_vector_lenghts.append(vectors[0])
12                  projection_vectors.append(vectors[1])
13                  projection_vector_lenghts.append(vectors[2])
14                  projection_vectors.append()
15             ENDIF
16        NEXT i
17        minimumIndex = minIndex(projection_vector_lenghts)
18        RETURN projection_vectors[minimumIndex]
19   ENDFUNCTION
```

Code 15: The SAT algorithm in pseudocode.

One thing that has to be added to the above algorithm is checking the axes perpendicular to the player's sprite's edges – since it is a square they are the x and y axis, with the normal vectors (1, 0) and (0, 1) accordingly.

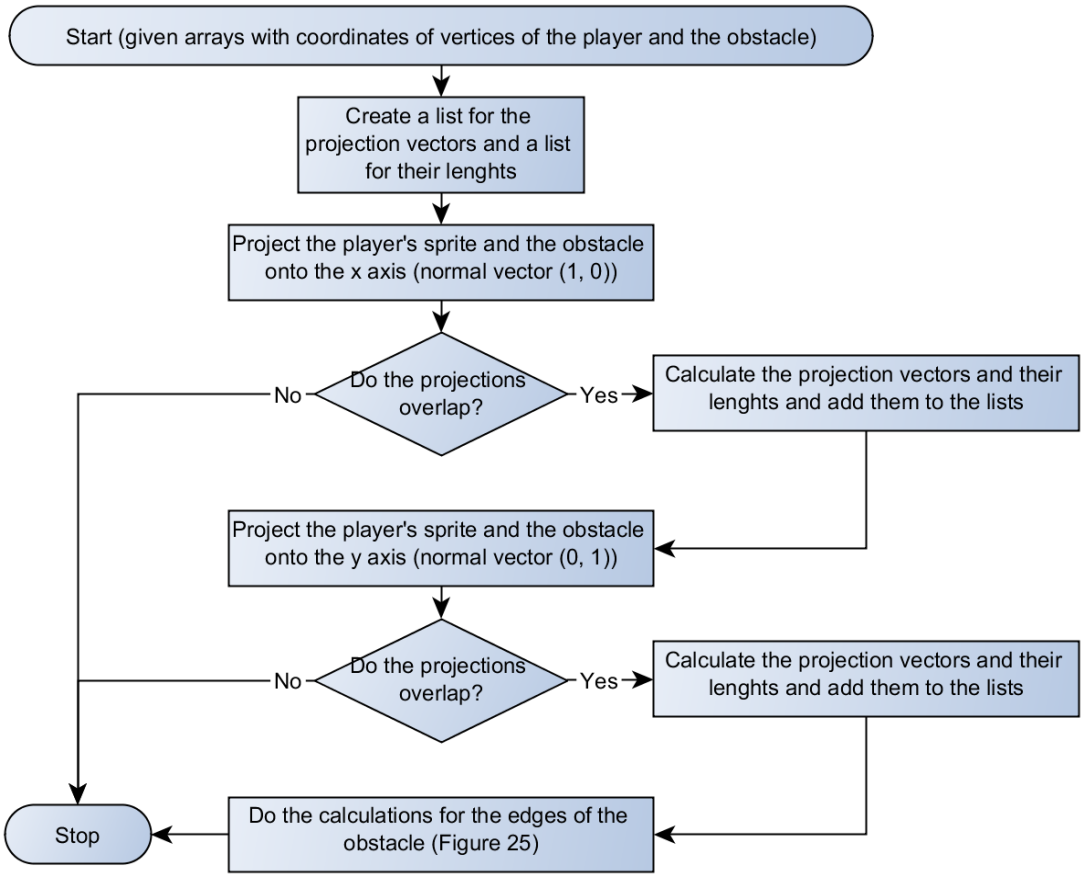


Figure 30: Checking the x and y axis. Note that in this scenario the projection vectors lists are created in the beginning and passed to the algorithm on Figure 29: The SAT algorithm., so that the choice in the end considers all the possible vectors.

```
1     FUNCTION collide(obstacle, player):
2           projection_vector_lenghts = []
3           projection_vectors = []
4
5           vectors = calculateProjectionVectors(obstacle, player, [1,0])
6           IF vectors = FALSE THEN
7                 // Thanks to the SAT we can break when there is no overlap on at
least one axis.
8                 RETURN [0,0]
9           ELSE
10                projection_vector_lenghts.append(vectors[0])
11                projection_vectors.append(vectors[1])
12                projection_vector_lenghts.append(vectors[2])
13                projection_vectors.append()
14          ENDIF
15
16          vectors = calculateProjectionVectors(obstacle, player, [0,1])
17          IF vectors = FALSE THEN
18                // Thanks to the SAT we can break when there is no overlap on at
least one axis.
19                RETURN [0,0]
20          ELSE
21                projection_vector_lenghts.append(vectors[0])
22                projection_vectors.append(vectors[1])
23                projection_vector_lenghts.append(vectors[2])
24                projection_vectors.append()
25          ENDIF
26
27          FOR i TO len(polygon) :
28                normal = getNormal(obstacle[i], obstacle[(i + 1) % len(obstacle)])
29                vectors = calculateProjectionVectors(obstacle, player, normal)
30                IF vectors = FALSE THEN
31                      RETURN [0,0]
32                ELSE
33                      projection_vector_lenghts.append(vectors[0])
34                      projection_vectors.append(vectors[1])
35                      projection_vector_lenghts.append(vectors[2])
36                      projection_vectors.append()
37                ENDIF
38          NEXT i
39          minimumIndex = minIndex(projection_vector_lenghts)
40          RETURN projection_vectors[minimumIndex]
41    ENDFUNCTION
```

Code 16: The SAT algorithm pseudocode updated to include the x and y axes.

The player's sprite

Now that we can successfully detect whether the player's sprite collides with an obstacle we need to make the player able to move the sprite around.

Player's sprite implementation (1.3)

The player's sprite will be a square. Its position on the screen will be stored as the x and y coordinates of its left top corner. pygame has functions that render a square of given width and height on screen at given coordinates, so this implementation is optimal.

Additionally, every time the player moves an array containing information about the sprite's vertices coordinates for the collision algorithm, so that it can be easily projected onto an axis.

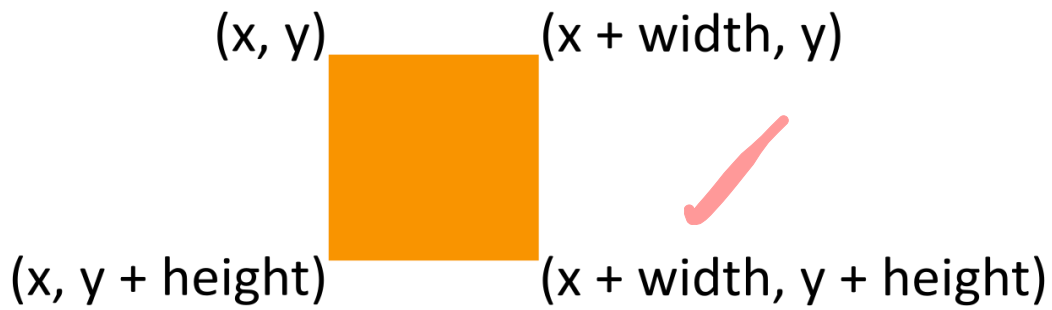


Figure 31: The coordinates of the sprite's vertices based on its x and y coordinates.

Navigation introduction

The user will be able to steer the sprite with the keys WSAD or the arrow keys, with the addition of space for jumping. This gives the user a lot of freedom to choose, though the preferable way of controlling the game is left hand on the WSAD keys, right hand on the mouse – the navigation is implemented in this way in the majority of modern games making the learning curve for my game less steep.

Making the navigation enjoyable is one of the most important things in a platform game since it is the thing that the user does the most. Therefore, the algorithm I devised for the navigation is quite complex, but it makes it smooth and intuitive.

The navigation algorithm (2.1)

pygame makes handling the user input easy by supplying the program with a set of events every time the game refreshes (approximately 60 times a second). It says what kind of an event was that (i.e. key pressed, key released) and details about each event (i.e. which key was that). By using this information it is possible to create complex steering mechanisms.

There are three characteristics of the player's sprite's movement that I want to include:

- When the user touches an obstacle and the up/w/space key is pressed, the sprite jumps.
- When a left/right/a/d key is pressed, the sprite starts to instantly move in the desired direction.

- When only one key is pressed, the sprite moves in the direction indicated by that key.

Making both the second and the third requirement to work is more complex than it initially seems since it needs to account for situations in which two buttons are pressed at a time, for example:

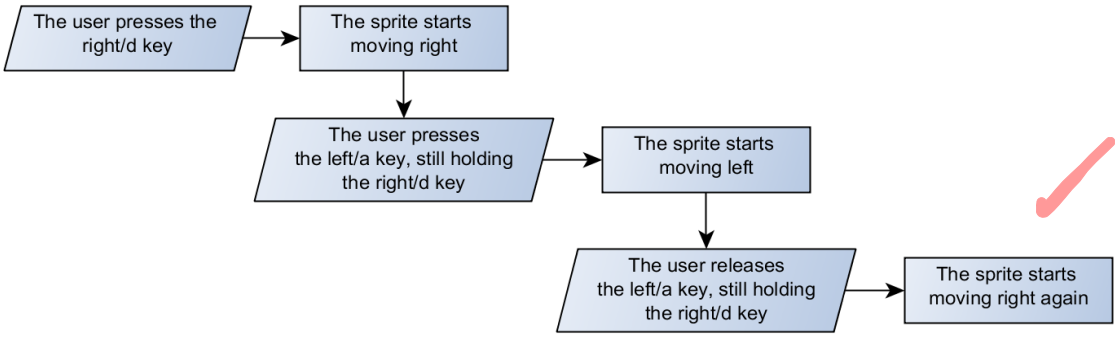


Figure 32: The situation that could possibly be confusing to the algorithm.

While solving that problem three observations are crucial:

- The changes of direction (or stopping) occur only when an event is received (a key is pressed or released).
- When a key is pressed the direction always is changed to the direction corresponding to the key.
- When a key is released the behaviour must depend on the state of the second directional key.

The solution to that is having two variables storing the current state of the left and right directional keys (including a and d) which get changed to one every time a button is pressed and to zero every time they are released. There should also be a variable storing the direction of the movement (-1 is left, 0 is no movement, 1 is right). It should be instantly updated after a key press is registered. When a key release is registered, it should be set to 0 if the second button's pressed variable is zero, or to the opposite direction if the second button's pressed variable is 1.

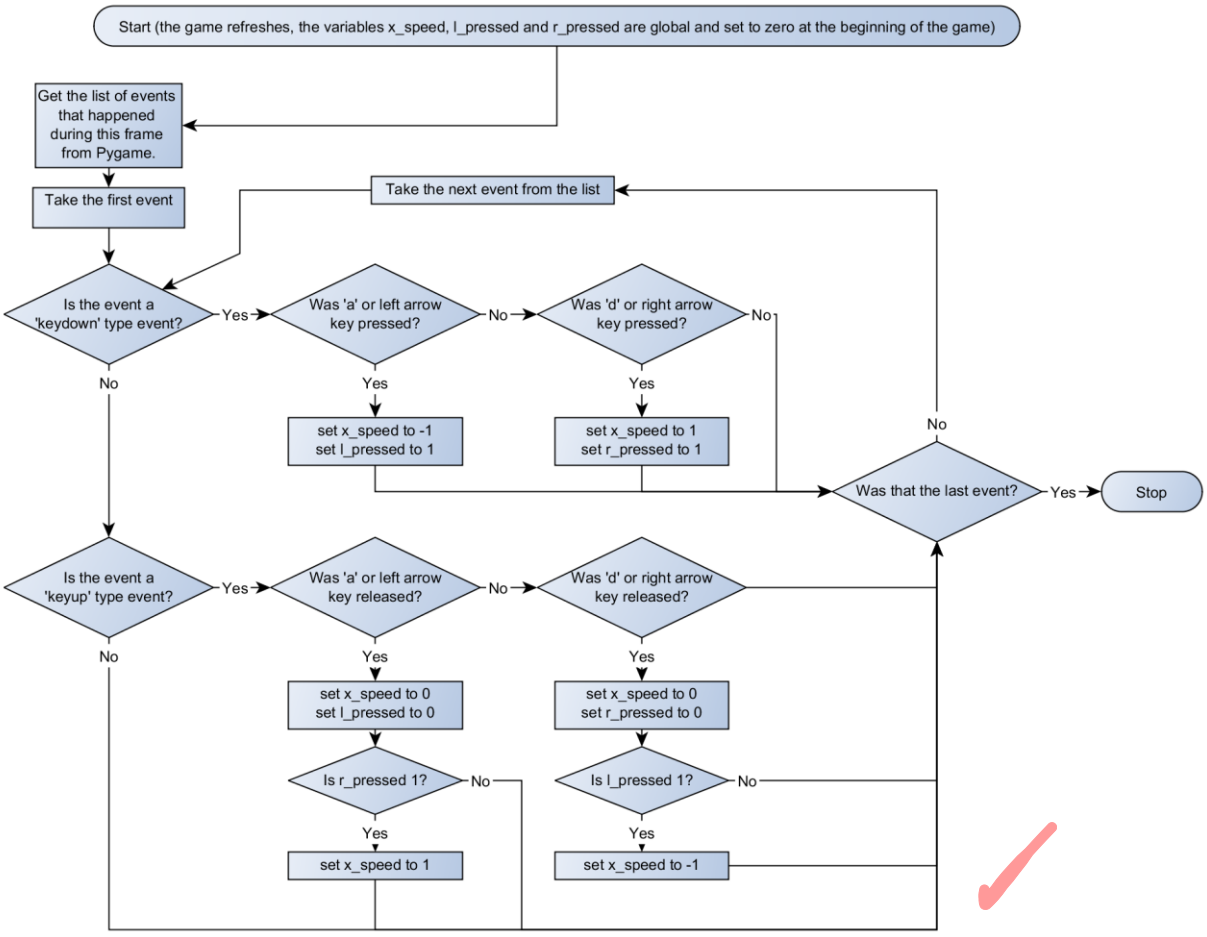


Figure 33: This algorithm takes a list of all the events that happened since the last game refresh and updates variables x_speed, l_pressed and r_pressed, which signify the direction of player's sprite's movement and the state of direction keys on the keyboard. x_speed can then be passed on to a function that will update the sprite's position.

```
FOR event IN pygame.listOfEvents:
    IF (event.type == "keydown") THEN
        IF (event.key == "a" OR event.key == "left") THEN
            left_pressed = 1
            x_speed = -1
        ELSE IF (event.key == "d" OR event.key == "right") THEN
            right_pressed = 1
            x_speed = 1
        ENDIF
    ELSE IF (event.type == "keyup"):
        IF (event.key == "a" OR event.key == "left") THEN
            left_pressed = 0
            x_speed = 0
            IF (right_pressed == 1) THEN
                x_speed = 1
            ENDIF
        ELSE IF (event.key == "d" OR event.key == "right") THEN
            right_pressed = 0
            x_speed = 0
            IF (left_pressed = 1) THEN
                x_speed = -1
            ENDIF
        ENDIF
    ENDIF
NEXT event
```

Code 17: The navigation algorithm written in Pseudocode.

event.type	event.key	left_pressed	right_pressed	x_speed
keydown	"d"	0	1	1
keydown	"a"	1	1	-1
mousemotion		1	1	-1
keydown	"g"	1	1	-1
keyup	"g"	1	1	-1
keyup	"a"	0	1	1
keyup	"d"	0	0	0
keydown	"left"	1	0	-1
keyup	"left"	0	0	0

Table 2: A trace table for the navigation algorithm. It demonstrates that it correctly works for situations when two keys are pressed at a time, as well as when other events (that should not influence the sprite's movement) are passed by pygame.

The above solution is significantly better from other considered by me previously, like checking the keyboard state every time (this would increase processing power required as the check will be performed every frame, and in the current solution it is done only when a change occurs), making the game consider a new input only after the previous key was released (might lead to confusion) or not doing anything when two buttons are pressed at the same time

(could lead to the movement being jagged and stopping when the user changes the buttons too fast).

The x_speed calculated by that algorithm can be added to the player's sprite x coordinate every frame, it can also be multiplied by a constant to adjust it for the gameplay needs.

Falling (4.9)

If the player's sprite is not "standing" on anything, it should be falling under gravity. To accomplished that I will make the player object hold a vertical velocity, to which a certain amount of speed will be added in every frame (just like in the real world, where the speed of a free-falling object changes around 9.81 ms^{-2} every second). After calculating the speed, it will be added to the player's sprite's y coordinate.

Jumping

Another important part of a platformer navigation system is jumping. It is crucial that the sprite jumps only when:

- It has just collided with an obstacle (it must be able to jump from something).
- The y component of the projection vector is bigger than 0 (the surface from which the sprite is pushing itself to jump up must be directed at least slightly upwards, it cannot be a vertical wall).
- Player is holding holding the up/w/space key.

The properties of the jump also must be specified. It should at least to a certain degree be similar to the real world or the player will find the mechanism unintuitive (i.e. I cannot just displace the sprite 30 pixels up in one frame and call it a jump). To achieve this I will utilize the mechanism specified in the Falling section – the vertical speed variable.

By setting the speed to a certain value directed up when the user jumps I can simulate the way it works in nature – when somebody jumps he increases his speed very fast, and then is slowed down in gravity. Similarly, in my game the player's y speed will be increased and the gravity simulating mechanism responsible for falling will make the jump smooth by slowly decreasing the speed and eventually making the object fall.

To get the necessary user's input we need to amend the flow chart and the pseudocode from The navigation algorithm section to include additional keys:

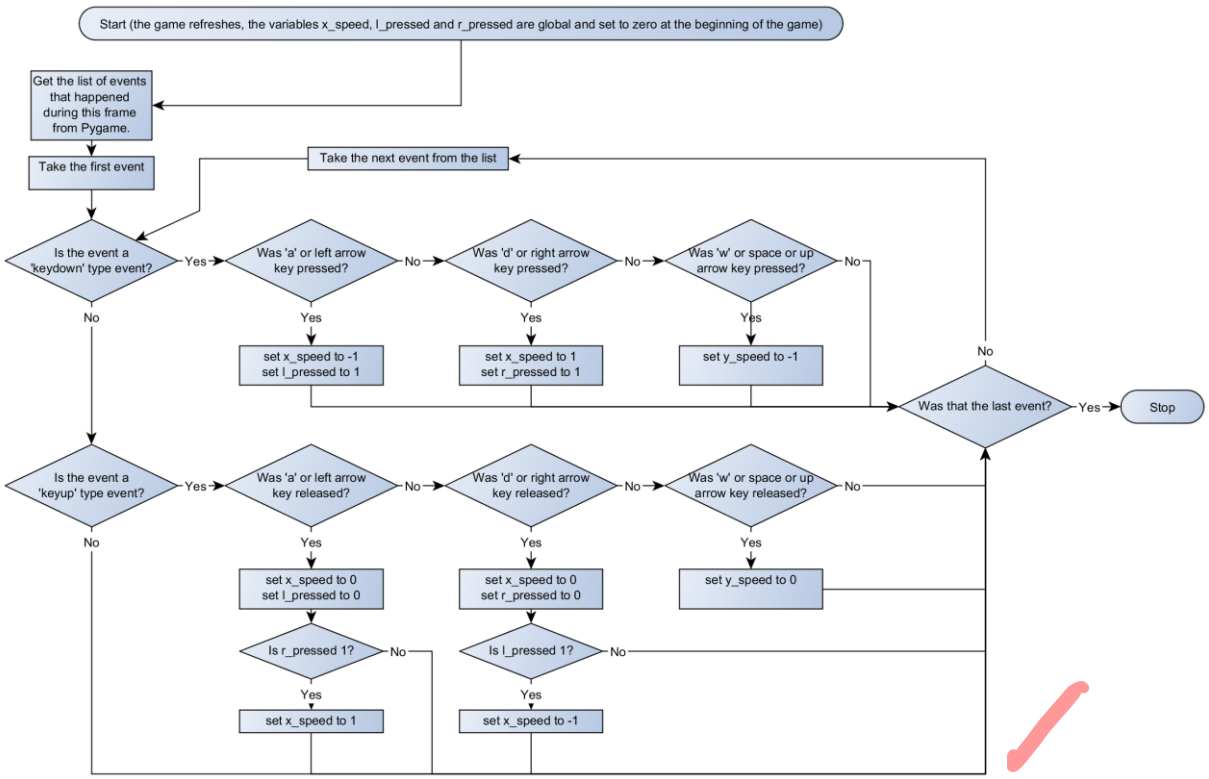


Figure 34: The navigation algorithm with jumping controls added.

```
01    FOR event IN pygame.listOfEvents:
02          IF (event.type == "keydown") THEN
03                IF (event.key == "a" OR event.key == "left") THEN
04                      left_pressed = 1
05                      x_speed = -1
06                ELSE IF (event.key == "d" OR event.key == "right") THEN
07                      right_pressed = 1
08                      x_speed = 1
09                ELSE IF (event.key == "w" OR event.key == "space" OR event.key ==
"up") THEN
10                      y_speed = -1
11                ENDIF
12          ELSE IF (event.type == "keyup") THEN
13                IF (event.key == "a" OR event.key == "left") THEN
14                      left_pressed = 0
15                      x_speed = 0
16                      IF (right_pressed == 1):
17                            x_speed = 1
18                      ENDIF
19                ELSE IF (event.key == "d" OR event.key == "right") THEN
20                      right_pressed = 0
21                      x_speed = 0
22                      IF (left_pressed = 1):
23                            x_speed = -1
24                      ENDIF
25                ELSE IF (event.key == "w" OR event.key == "space" OR event.key ==
"up") THEN
26                      y_speed = 0
27                ENDIF
28          ENDIF
29    NEXT event
```

Code 18: The pseudocode for the navigation algorithm with jumping inputs added.

Notice that the y_speed is set to -1. This is because pygame considers the top left corner of the window as the origin of the coordinate system, with the positive direction of the y axis pointing down. Therefore, if we want to make the sprite move upwards we need to make its speed negative.

Now that we know whether the user wants to jump we need to perform the jump itself, first checking if the conditions are met:

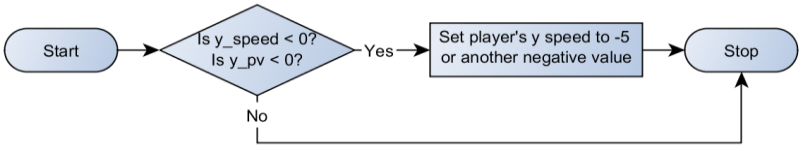


Figure 35: The check whether the sprite should jump. y_pv is the y component of the projection vector.

```
1    IF (y_speed < 0 AND y_pv < 0) THEN
2          player.y_speed = -5
3    ENDIF
```

Code 19: The pseudocode representing the check performed to determine whether the user wants to jump and whether he should be allowed to do so.

Notice that we only need to check whether y_p is smaller than zero (directed up) since if there was no collision it will be 0.

Displacement after collision (4.1)

After the player makes his move the collision algorithm (The Separating Axis Theorem) checks for overlap with obstacles and returns a collision vector, a vector by which the player's sprite should be displaced to get out of the collision. So to get out of the collision we just need to add the x and y components of the projection vector to the player's sprite's x and y coordinates.

The complete player navigation loop

When the game refreshes, we should first update the level based on the mouse position, as detailed in the Rendering the level section. Then based on the keyboard input we update the player's position, remembering to take into account gravitational change in the y component of the speed (the Falling section). As soon as the player makes his move we check for collisions and calculate the projection vector, by which the sprite is then displaced.

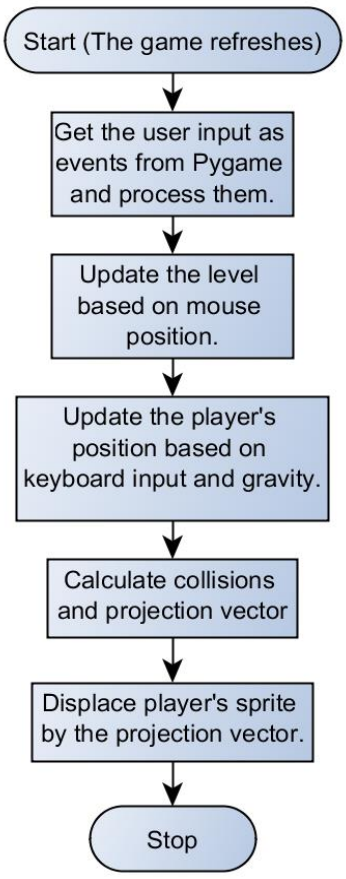


Figure 36: The process of updating the level and player that happens every time the game refreshes.

more elements of
usability explained

User Interface

While the gameplay is the most important element of the game, the user interface is also important for the enjoyment of the user and ease of use, and therefore can be considered a crucial usability feature.

Design of the main screen (1.8, 1.12, 3.1)

The main screen is the first thing the user sees when he enters the game. It should introduce the game's title, as well as clearly show the available levels with high scores obtained. The first design I came up with is shown below.

Figure 37: The design of the main screen. I created it in Inkscape (an open-source vector graphics programme) ready to be used in-game since my Inkscape drawing skills are better than normal drawing skills, and the geometric design I came up with was easier to execute in a vector graphics programme.

The design incorporates the information about the locked and unlocked levels (the greyed-out colours and lock icons), and the number of stars obtained is displayed. The logo seen at the top of the screen is a simple one (as indicated in the interview, designing the logo is not a priority) and consists of the title of the game[5] and a simple graphic presenting a classic example of a dimension manipulation trick – a two-dimensional square surrounded by a circle on a plane cannot get out of its prison, but once we give it access to the third dimension, jumping above the obstacle is simple. This is a metaphor for the game mechanic, which uses this concept in a more complex way. The background is populated with polygons representing

[5] The font used for the logo is *Lobster*, which is licensed on the Open Font License (http://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL_web), meaning that it can be used in non-commercial products without attribution and with attribution on commercial products. This makes it perfectly suitable for this non-commercial project, and if I were to sell the game in the future, a simple attribution will enable me to continue using the font.

the mathematical character of the game and some shadows are added to create illusion of depth hinting at the fact that the game extends beyond what we see on screen.

The advantage of using a computer graphics program for design is that the design can be separated into layers for easier rendering:

Figure 38: The image on the left is the base layer. Star scores and locked level badges can be rendered onto it by pygame.

Since the main screen is already designed I must now concentrate on bringing it into life and tying it into the game.

The game's lifecycle

There are three states in which the game might possibly be: the main screen, a level, and a screen informing the player that he has beaten the level. The game needs to know the state in which it currently is at all times to render appropriate screen elements and properly react to events. Thus, a variable must be created that will store the current state.

The states should be assigned numbers so that they are easily identifiable. Additionally, this will enable me to store the current level number in the same variable. The numbers assigned to each state will be:

- 0 – main screen
- -1 – winning screen
- 1, 2, 3, … - levels

By storing the non-level states at the beginning of the list it is easy to assign the levels numbers that match the level numbers displayed to the user, making the implementation easier. It can also be easily checked whether the game is in a level by comparing the state number to zero (if it is bigger, then we are in a level).

The lifecycle of the game will thus look like this:

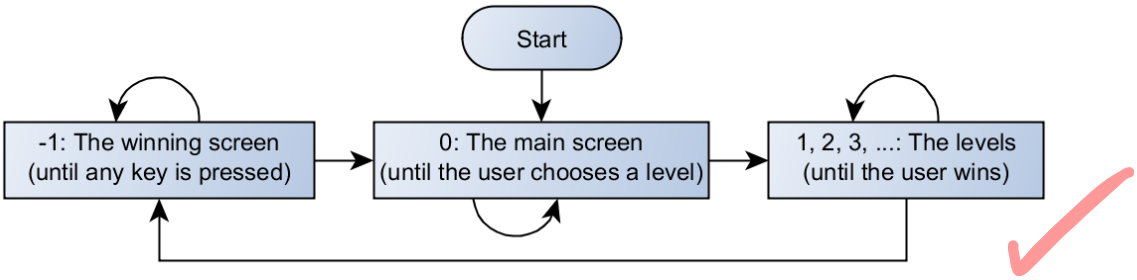


Figure 39: First draft of the game's lifecycle. The arrows directed in a loop signify the fact that the game is refreshed around 60 frames per second, so every state will be processed with every refresh until the user acts.

To implement this model, every time the game refreshes it will check in which state it currently is:

```
1     IF state == -1 THEN
2           // Process and render the winning screen
3     ELSE IF state == 0 THEN
4           // Process and render the main screen
5     ELSE IF state > 0 THEN
6           // Process and render the level indicated by the state variabel
7     ENDIF
```

Code 20: This pseudocode fragment shows the check which will be performed with every refresh of the game.

Thanks to this model changing the state of the game should be as simple as changing a single variable.

Rendering the elements of the main screen (3.1)

The background of the main screen is pre-rendered as noted in the Design of the main screen section, what the programme needs to do is add star scores and blocked badges to the level icons. The dimensions shown below will be the reference used during rendering:

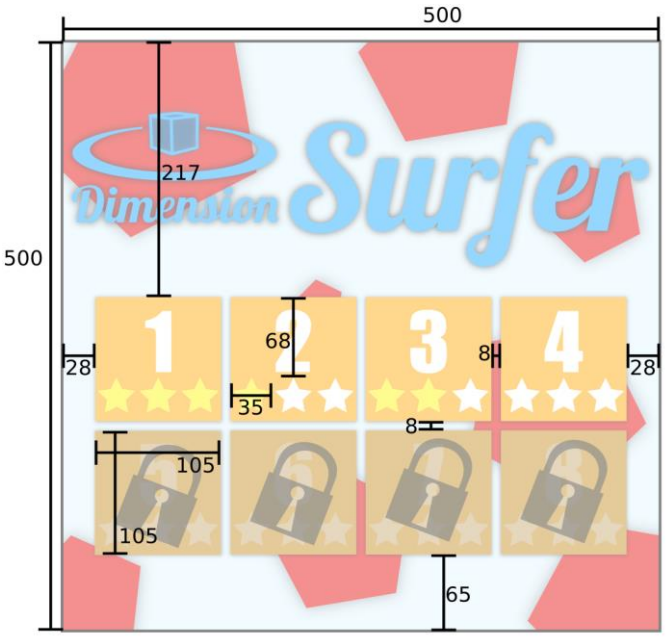


Figure 40: A reference image with the dimensions of the main screen in pixels.

The background image can simply be loaded from a bitmap and rendered, whereas the stars and the locked badges must be additionally positioned.

To render the information properly the game has to access the high scores information saved in the file described in Saving high scores. The information saved in the file can be split into an array using the spacebar as a delimiter. Then by looping through it we can get the information about each level and display appropriate data (number of stars, locked badge).

The star rendering function was described in Rendering the star score, it just needs to be supplied with a star bitmap of correct size. The locked badge is just a square bitmap which needs to be rendered at correct height.

As can be seen on Figure 40: A reference image with the dimensions of the main screen in pixels., the first row of stars starts at x coordinate 28 and y coordinate 285. The next one will be 113 pixels to its right (width of the tile and the gap), the next one will again be further 113 pixels to the right and so on. The stars in the lower row are 113px lower than the ones in the upper column. Knowing that we can use the modulo and integer division operations to compute the coordinates of the stars based on the level number.

An important thing to remember is that the levels are indexed from one up and for the modulo operation to work properly they need to be indexed from 0.

$$x = 28 + \big((n-1)\ \%\ 4\big) * 113$$
$$y = 285 + ((n-1)\ //\ 4) * 113$$

The % sign is a symbol for the modulo operation and // is a symbol for integer division (a method of division that rejects the remainder).

To check whether the method works I will use a trace table:

n	x	y
1	28	285
2	141	285
3	254	285
4	367	285
5	28	398
7	254	398

It can be seen that the function does give correct coordinates for different indexes of the levels. Now the same function has to be implemented for the locked level badges. The problem is basically the same, we just need to change the starting coordinates:

$$x = 28 + \big((n-1)\ \%\ 4\big) * 113$$
$$y = 217 + ((n-1)\ //\ 4) * 113$$

Letting the user choose a level (2.3)

Now that we have a properly rendered main screen we need to detect the intentions of the user, namely – let him choose a desired level. pygame lest me detect mouse clicks in a way

similar to the one described in The navigation algorithm section, but the event type is now *mousebuttondown*. Then by getting the coordinates of the pointer and doing some calculations with them I can get the number of the level.

To get the number of the level I need to calculate in which column and in which row the mouse pointer is at the moment. First I take the x and y coordinates and subtract from them the coordinates of the first level box to get mouse coordinates relative to the level-choosing part of the screen). To get the column I can take the relative x coordinate and divide it by the width of one level tile neglecting the remainder. To get the row I need to do the same with the relative y coordinate. In both cases the indexing starts at zero. There are four level tiles in each row, so for each row I need to add four to the level number. Then by adding the column number to that and adding an additional one (level indexing starts with one) I get the level index.

$$level\ index = 4 * \Big((mouse_y - 213)//113\Big) + (mouse_x - 24)//113 + 1$$

// again symbolises integer division. Note that we add a border of 4px to every tile that is also clickable to cover the empty spaces between the tiles.

An important thing to note is that the above equation will calculate the level index correctly only if the user clicks an actual level tile – clicking for example below the tiles will add an additional (non-existent) column to the calculation, giving a level index bigger than normally possible. Therefore, it is crucial to perform simple validation on the mouse coordinates: check whether they are within the box containing the tiles.

The above is summarized in the following flow chart:

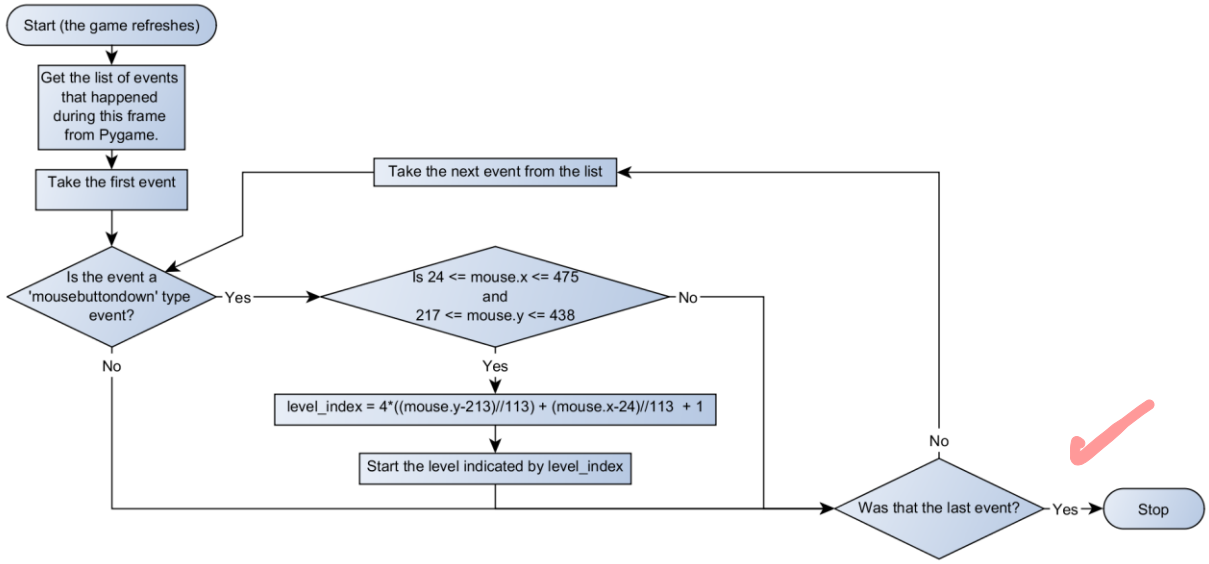


Figure 41: The level choice event loop. mouse.x and mouse.y are the mouse cursor coordinates.

And this is the corresponding pseudocode:

```
1     FOR event IN pygame.listOfEvents:
2           IF (event.type == "mousebuttondown") THEN
3                 mouse = getMousePosition();
4                 IF (24 <= mouse.x AND mouse.x <=475 AND 217 <= mouse.y AND
mouse.y <= 438) THEN
5                       level_index = 4*((mouse.y-213)//113) + (mouse.x-24)//113  + 1
6                       startLevel(level_index)
7                 ENDIF
8           ENDIF
9     NEXT event
```

Code 21: The level choosing algorithm in pseudocode.

Tutorial (1.10, 1.11)

It is important to introduce the user to the basic concepts of the game. The best way to do this is by introducing the game concepts during the gameplay. One of the best examples of that is the first level of Super Mario Bros., mentioned in the section.

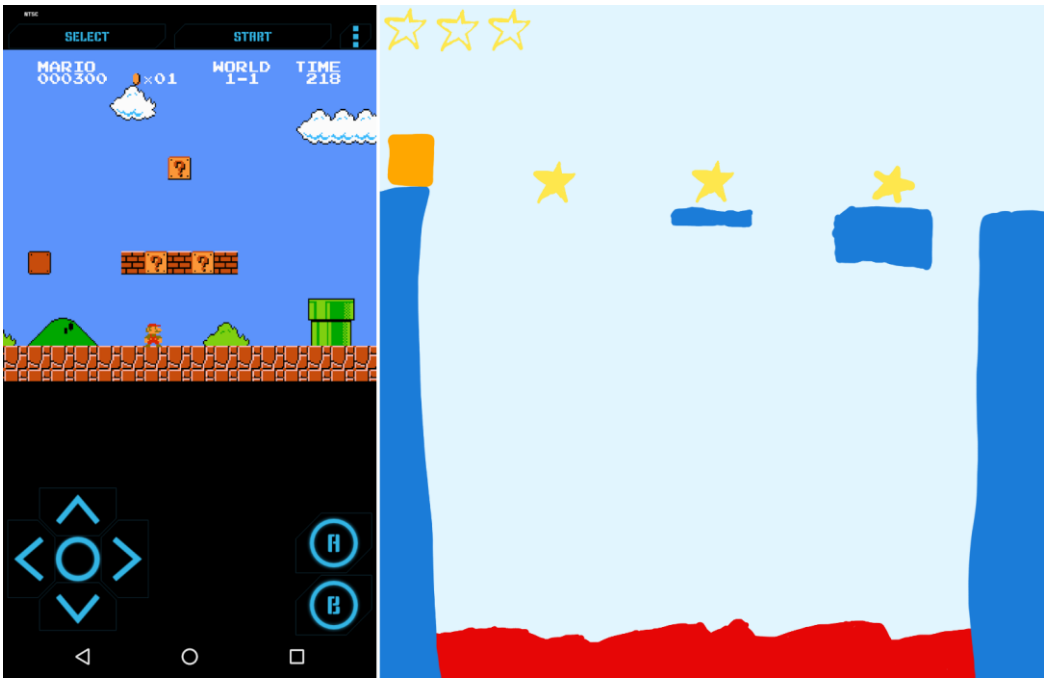


Figure 42: The first level of Super Mario Bros. and the first level I designed for my game.

The beginning of the first level of Super Mario Bros. swiftly introduces the player to the basic concepts of the game without displaying any on-screen help. The player learns that the objective of the game is to go from left to right, that there are platforms to jump on, enemies to avoid and power-ups to collect.

I tried to apply the same principles when designing the first level. It introduces the player to the navigation system and the goal (get to the other side is a clear message in this situation), the stars (there is no way to finish the level without collecting them), the lava surfaces (the player will probably die in a few of the first attempts) and the idea of being able to manipulate the look of the level (it is impossible to complete it without making some platforms appear).

I decided to add two additional information screens – one describing using the wsad keys to navigate, and another one, animated, to introduce the idea of third dimension manipulation. These will appear at the beginning of the first level and will be dismissible by a mouse click.

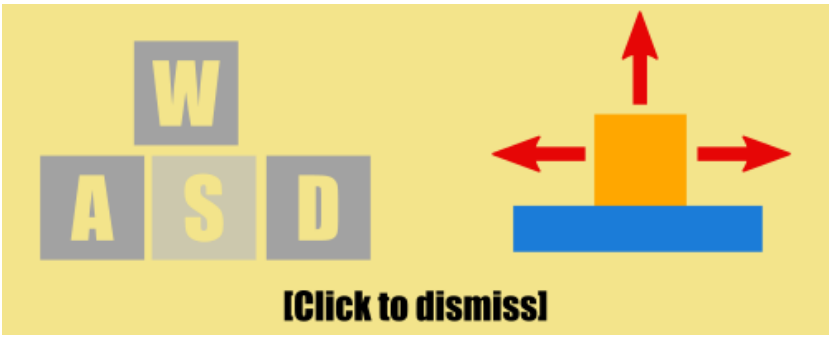


Figure 43: The basic navigation description image – the keys w, s and d are shown to be the ones to use for navigating.

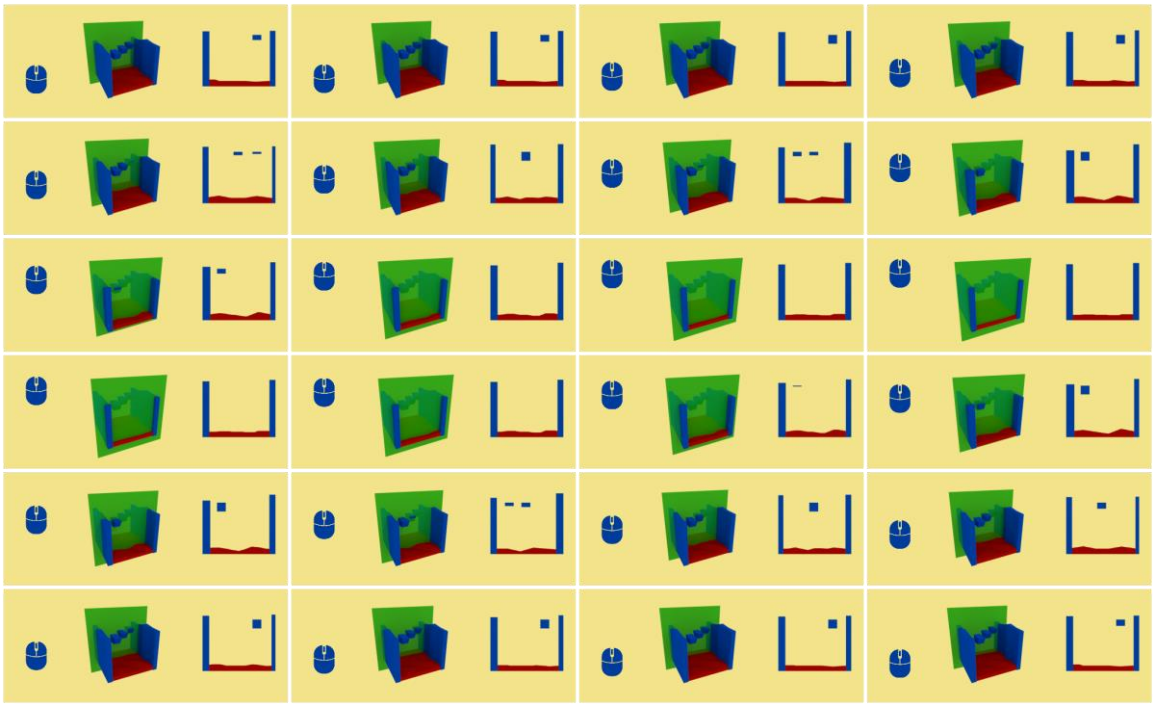


Figure 44: Frames of the animation describing the third-dimension manipulation mechanism. The blue mouse on the left side of the frame moves up and down (indicating the user action required) and as a result the green cutting plane moves through the 3D model. This is synchronized with the changes of the level image. This should help the user create a mental link between the changes in the level geometry and an imaginary plane moving through the three-dimensional environment.

Displaying the tutorial (1.10)

Three things that are required for the tutorial have to be accomplished:

- Display it only at the beginning of the game.
- Allow the user to dismiss a tutorial screen with a mouse click.
- Animate the second tutorial screen.

Whether the user is playing the first level can be easily checked with the 'state' variable introduced in The game's lifecycle section. pygame's event loop allows me to detect mouse clicks easily. The challenge of animating the tutorial is more complex – it is designed to be

played at around 6 frames per second and the game refreshes at a rate of around 60 frames per second. Therefore, a new frame of the animation will have to be displayed every 10 frames of the game. I can store the actual game frame in a variable and obtain the frame to be rendered using integer division and modulo:

$$frame_{to\ render} = (frame_{game}\ //\ 10)\ \%\ 24$$

The pygame event loop was explained in detail in The navigation algorithm section, so in this section its details will be omitted. Detecting mouse clicks is a matter of adding an additional condition checking whether the type of event is *MOUSEBUTTONDOWN*.

A variable holding the state of the tutorial will be introduced, zero indicating that the tutorial should be displayed, one that the first screen is shown, and two that the animated second screen is being shown. Anything bigger than that will indicate that the tutorial has been dismissed.

The state variable should be set to zero when the game is started. A *draw()* function will be called every frame and will render the appropriate tutorial screen as indicated by the state variable. When a mouse button is clicked, *nextTutorial()* will be called, which will increase the state variable by one.

Testing

The criteria used for testing will be similar during the development and the evaluation stage. To test the game, I will be playing it extensively to check all the possible scenarios and functions. Some of the features that will have to be tested are:

- Is the player able to choose the correct level? (tested by attempting to choose different levels)
- Are the scores and locked badges displayed correctly? (proper values and correct placement)
- Are the collisions and projection vectors calculated accurately and in a seemingly natural way? (this can be tested both by printing the values and by observing the behaviour of the sprite during gameplay. It must feel 'natural', behave exactly as the player expects it to behave).
- Does the player die when collided with a lava surface? (tested by falling into lava repeatedly)
- Are the stars collectible? Is the calculated star score always correct? (tested by collecting stars)
- Are the constants used when jumping and moving selected properly so that navigation is intuitive and makes the levels possible to win? (again, this is subjective and must 'feel right')
- Are the levels possible to beat? (tested by playing the levels myself and giving them to the test user)

All the features that the game should have are described in detail in Requirement specification (success criteria) and I will use this list to fully test the game's working.

Robustness testing also needs to be performed. This means providing the game with unexpected inputs and behaving in unnormal ways to try to confuse it. Behaviours like this include:

- Clicking the mouse in random places on the main screen.
- Trying to choose locked levels.
- Pressing random keys during gameplay.
- Moving mouse around very quickly during gameplay.
- Beating all the levels.
- Switching focus to another window.
- Trying to play the game on a very slow computer.

Additionally, during the development process I will have to extensively test the workings of the Separating Axis Theorem. To test it in the most versatile way possible I plan to implement two testing mechanisms that will give me a bigger degree of control than is given to a player of the final game. To achieve that, I will make the player's sprite follow the mouse's cursor so that it can be placed anywhere around the obstacle. This can be represented in pseudocode:

```
1    mouse = getMousePosition()
2    player.x = mouse.x
3    player.y = mouse.y
```

Code 22: Pseudocode for making the player's sprite follow the mouse.

I will also use abstract geometric figures instead of normal levels during the implementation stage to see whether the collision system can cope with complicated collision situations.

Design 15/15 This is an extremely detailed design stage with problems broken down into computational solutions and justified. Full algorithms are given in the form of flowcharts and a textual description as well as relevant trace tables.
There is a lengthy explanation of the usability features and variables/classes in addition to testing strategies which are carried out in the implementation and evaluation stages.

Implementation

Now that the design section is ready I can start the implementation. First, I will lay out the programming environment that I used.

The programming setup

I used the PyCharm Community Version[6] IDE for programming in Python as it has multiple features that make the code writing and editing faster and more enjoyable. It also has easy to use debugging features, which has the potential to help while programming the project.

I used Git, which is a popular version control system, for keeping track of the changes I made in the code. Since I planned for the implementation to be iterative and to build up on the code, sometimes changing parts of it, I anticipated that having all the changes stored and neatly organised will benefit the process greatly. I also used GitHub, which is a Git service partly placed in the cloud, allowing me to access and review the code online, as well as provided me with a desktop client which greatly simplifies the usage of Git.

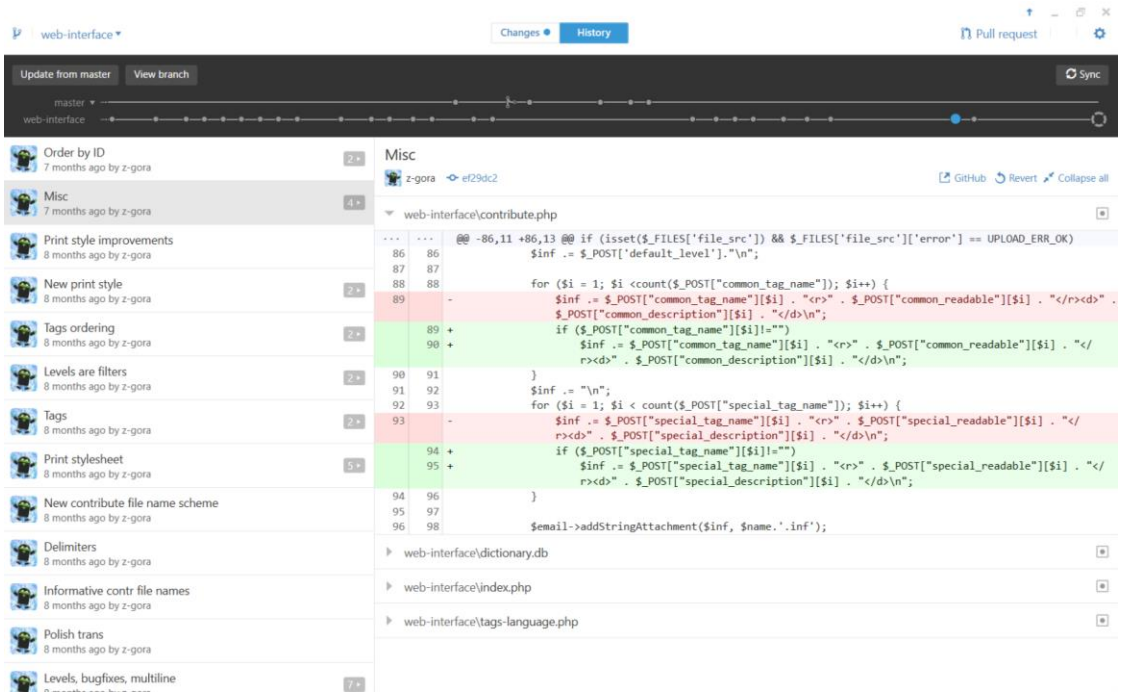


Figure 45: The GitHub app interface showing the repository of another programming project I have done in the past. The screenshot shows some of the main features of Git – a commit (changes) history is on the left, and each commit has all the changes in code (the right section, deletions in red, additions in green) outlined. I can also revert the code to any of the commits.

The software I used for 3D modelling, Blender, has got a built-in Python scripting engine that allows me to write code and run it easily on a given 3D model.

[6] https://www.jetbrains.com/pycharm/

Exporting the levels (1.1, 4.6)

To export the level I first must create it. For testing purposes, I decided to go for a few so called icospheres – not playable, but very complicated in shape and quite distinctive, so perfect for testing the export and rendering mechanism. Also, all their cross sections are convex shapes, which is required by the SAT.

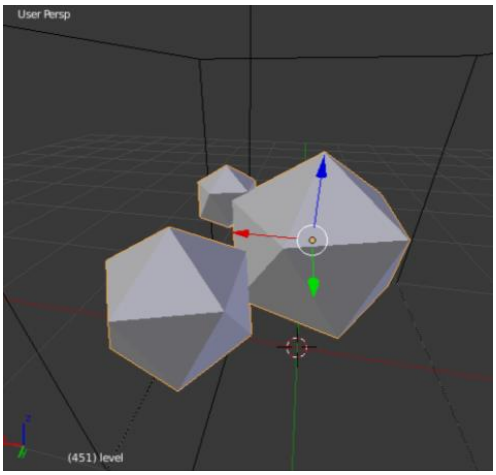


Figure 46: The icospheres.

Based on the algorithm devised in the design section and the Blender Python scripting library documentation I created this Python code:

```
1.  # Import the Blender scripting library.
2.  import bpy
3.
4.  # Map is a function used to translate Bledner coordinates
5.  # to the ones used by Pygame.
6.  # It accounts for different scales and origin positions.
7.  def map(number):
8.      return (number+1)*250
9.
10. # Create the file in which the data will be stored.
11. with open('1_level.txt', 'w') as f:
12.     # There will be 500 cross-sections, since the game window
13.     # will be 500x500 pixels and I want the mouse's cursor position
14.     # to represent a cross-section.
15.     for i in range(1,501):
16.         # Move the cutting plane to the correct position.
17.         bpy.data.objects['cut'].location[0] = 2/500*i-1
18.         # Select level object...
19.         level_main = bpy.data.objects['level']
20.         level_main.select = True
21.         bpy.context.scene.objects.active = level_main
22.         # Duplicate it... (we will be working on a copy)
23.         bpy.ops.object.duplicate()
24.         # And select the copy.
25.         level = bpy.data.objects['level.001']
26.         level.select = True
27.         bpy.context.scene.objects.active = level
28.         # Then add and apply the modifier.
29.         bpy.ops.object.modifier_add(type="BOOLEAN")
30.         level.modifiers["Boolean"].object = bpy.data.objects['cut']
31.         level.modifiers["Boolean"].solver = "CARVE"
32.         bpy.ops.object.modifier_apply(apply_as='DATA', modifier="Boolean")
33.
```

```
        # Export the data:
        # Iterate over all the cut objects faces.
        for face in level.data.polygons:
            # Create a list for holding the vertices of the face.
            faceList = []
            # Iterate over the vertices of the face.
            for vertID in face.vertices:
                # Transform Blender's own Vector object into a tuple.
                vectorVert = level.data.vertices[vertID].co.to_tuple()
                # If the vertex is on the cutting plane,
                # append to faceList.
                if (round(vectorVert[0],5) ==
                round(bpy.data.objects['cut'].location[0],5)):
                    faceList.append([map(vectorVert[1]),
                    map(vectorVert[2]*(-1))])
            # If the obtained face is a polygon (more than two vertices),
            # add it to the data file.
            if (len(faceList) > 2):
                for vec in faceList:
                    f.write(str(vec[0])+" "+str(vec[1])+"\n")
                f.write("\n")
        # Remove the copy of the level object
        bpy.ops.object.delete()
        # As specified, a '#' symbol signifies end of data
        # about a cross-section
        f.write("#\n")
# Close the file
f.closed
```

Code 23: The exporting algorithm.

After running it with the model I did obtain the file containing the data about polygons in each dimension:

```
...
349.35732185840607 250.3225803375244
335.349977016449 320.07745653390884
298.67778718471527 346.7216342687607
201.32221281528473 346.7216342687607
164.65002298355103 320.07745653390884
150.64267814159393 250.3225803375244
168.8939407467842 194.148451089859
250.00000054243958 157.72873908281326
331.1060592532158 194.148451089859

#
403.3983498811722 298.06336015462875
402.92416512966156 299.5228134095669
402.54051983356476 299.69508200883865
402.15687453746796 299.5228134095669
401.68270468711853 298.06336015462875

348.63077849149704 252.3225862532854
335.02504974603653 320.07745653390884
299.4043231010437 345.9576889872551
200.5956768989563 345.9576889872551
164.97495025396347 320.07744908332825
151.36922150850296 252.3225862532854
```

```
170.27029395103455 194.148451089859
250.00000054243964 158.3467721939087
329.72970604896545 194.148451089859

#
...
```

Code 24: A sample of the contents of the level data file.

To export the lava surface, I created a second 3D mesh called 'lava' and amended Code 23 to refer to this object and to create a second data file containing information about the lava polygons.

```
1.  ...
11. with open('1_lava.txt', 'w') as f:
12. ...
19.         level_main = bpy.data.objects['level']
20. ...
25.         level = bpy.data.objects['level.001']
26. ...
```

Code 25: Amendments to Code 23 so that it creates a second file containing data about the lava surfaces.

I added the new version at the end of the Python script file so that when the script is run both the data about the level and about the lava surfaces are exported. Now the only thing left to do was to add a third file with the position of the stars. I decided to create it within the Blender Python script and then manually add the stars coordinates. The file is created if it does not already exist.

```
107.    # Create the stars data file if it does not exist
108.    open("1_stars.txt", 'w').close()
```

Code 26: Creating the stars data file. Python attempts to open it (creating the file if it does not exist, and then closes it without writing any data.

The core pygame code

To use the pygame library a basic code needs to be set up. This will be the skeleton of the game:

```
1.  # Import the pygame library and the math library, which will be useful later.
2.  import pygame
3.  import math
4.
5.  def main():
6.      # Initialize the pygame environment.
7.      pygame.init()
8.
9.      # Set the width and height of the screen.
10.     size = (500, 500)
11.     screen = pygame.display.set_mode(size)
12.
13.     # Set the title of the window.
14.     pygame.display.set_caption("My Game")
15.
16.     # This variable stores whether the user pressed the close button on the window.

17.     done = False
```

```
18.
19.     # An object used to manage how fast the screen updates.
20.     clock = pygame.time.Clock()
21.
22.     # Main program loop, runs until the close button is pressed.
23.     while not done:
24.         # Event processing - we iterate on the events given to us by pygame:
25.         for event in pygame.event.get():
26.             # If the event type is QUIT, the user wants to close the window.
27.             # So we set done to True.
28.             if event.type == pygame.QUIT:
29.                 done = True
30.
31.         # Game logic:
32.
33.         # Do the drawing:
34.         screen.fill((255, 255, 255))
35.
36.         # Update the screen:
37.         pygame.display.flip()
38.
39.         # Show the frame rate in the title for performance checking.
40.         pygame.display.set_caption(str(clock.get_fps()))
41.         # Set the desired frame rate to 60fps (frames per second.
42.         clock.tick(60)
43.
44.     # Close the window when the main loop finishes.
45.     pygame.quit()
46.
47. if __name__ == "__main__":
48.     main()
```

Code 27: The game's skeleton.

The main loop is the part of code responsible for refreshing the game window. It consists of four stages: processing events, doing the necessary calculations (game logic), drawing (rendering) and updating the screen. The main loop is stopped when the window is closed: the *done* variable is set to True and the while loop does not run again, making line 44 execute and the program closes.

Rendering the level

Now that we have the level data exported from Blender and the core pygame code written, we can start rendering the levels. We will start with creating a class called *ThreeDMesh* which will handle the data import, mouse position reading and drawing the polygons. Then I will be able to create two new classes that will inherit from the *ThreeDMesh* class – *Level* and *Lava*, which are both rendered in the same way, but handle collisions differently.

The class would have to store the id of the object (number of the level), the polygon data, position in the third dimension (z), the base colour, the darker version of it and the current colour. The methods of the class will have to be used to import the polygon data, update the object when the user changes the mouse position, and finally draw the polygons every frame.

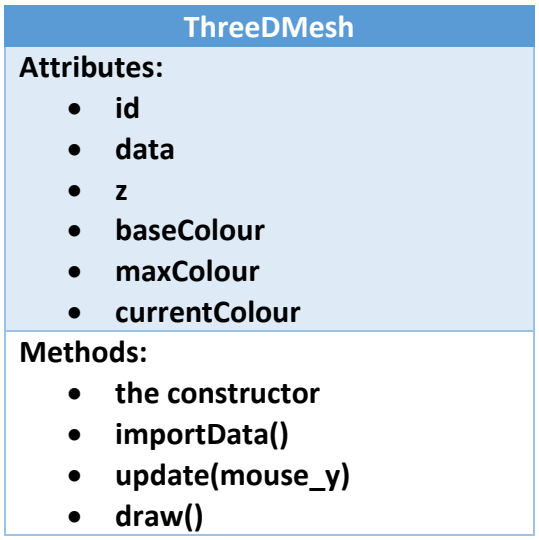


Figure 47: The class diagram of the 3Dmesh class.

First, let's write the constructor:

```
1.  class ThreeDMesh():
2.      def __init__(self, id, baseColour, maxColour):
3.          self.id = id
4.          self.data = self.importData()
5.          self.z = 0
6.          self.baseColour = baseColour
7.          self.maxColour = maxColour
8.          self.currentColour = baseColour
9.
10.     # Other methods will go here...
```

Code 28: The constructor of ThreeDMesh.

Now I can concentrate on writing the other methods that will be in the ThreeDMesh class.

Importing data from the text files (4.6)

The algorithm for doing that was devised in the Design section and proved to work. Translated into Python it becomes:

```
10.     # This method will import polygon data from a text file.
11.     def importData(self):
12.         # Create an empty list for the data.
13.         data = [[[]]]
14.         # Create indexes that will help iterate over polygons
15.         # and the third dimension.
16.         polygonIndex = 0
17.         zIndex = 0
18.         # Open file specified by the id.
19.         with open('level_data/' + self.id + ".txt", 'r') as f:
20.             # Parse every line in the file.
21.             for line in f:
22.                 if line == "#\n":
23.                     # If line contains a '#' it means that the data about a
24.                     # particular cross section has just finished. Create a new
25.                     # sublist for the next cross section.
26.                     data[zIndex].pop()
27.                     data.append([[]])
28.                     # We move to the next cross section, so we increase the zIndex.
29.                     zIndex += 1
```

```
                    # We have not added any polygons to this cross section yet,
                    # so we reset the polygonIndex to 0.
                    polygonIndex = 0
                elif line == "\n":
                    # If the line is emty, that means that we reached the end of
                    # a single polygon's description. We create a sublist to hold
                    # the next polygon...
                    data[zIndex].append([])
                    # ...and change the polygonIndex to indicate that we moved to
                    # the next polygon.
                    polygonIndex += 1
                else:
                    # If the line is neither empty nor it has a '#' in it,
                    # we assume that it contains coordinates of a point.
                    # We add this point to a polygon indicated by polygonIndex
                    # in the crossSection indicated by the zIndex.
                    data[zIndex][polygonIndex].append([float(x) for x in line.split(
    " ")])
            # The pop() functions throughout the code are to get rid of unpopulated
            # lists that occur naturally due to the construction of the data format.
            data.pop()
        # Close the file...
        f.closed
        # ...and return the processed data array.
        return data

```

plenty of annotation of code throughout

Code 29: The method that imports data about a mesh from a file specified by the id.

The output of this method is stored in *self.data*, as seen in the constructor. Note that the data is stored in a folder *level_data* in the game's directory.

Drawing a single cross section (3.4)

Now we can implement the drawing of a single cross section using pygame's *draw.polygon()* function, which takes the drawing surface, drawing colour and a list with the coordinates of the desired polygon's vertices. Thanks to the data list structure, we can just iterate on sublists of a give cross-section sublists, and each of them can be just passed to the pygame's function to be drawn.

As noted in Improving the controls (in the Rendering the level section), we cannot just use the attribute *z* of our object as a list index since it will not always be an integer. We round it to a integer and use this instead.

```
# This method will draw a cross-section specified by self.z
    def draw(self, screen):
        # This is a set of polygons in the cross-section that
        # we will be drawing:
        drawing = self.data[math.floor(self.z)]
        # We iterate on the elements of the drawing list,
        # which are lists of vertices...
        for polygon in drawing:
            # ...and pass them to the drawing function.
            pygame.draw.polygon(screen, self.currentColour, polygon)

```

Code 30: The method that draws a single cross-section.

To actually draw the polygons we have to instantiate the class and then call the *draw()* method in the main game loop. We need to write

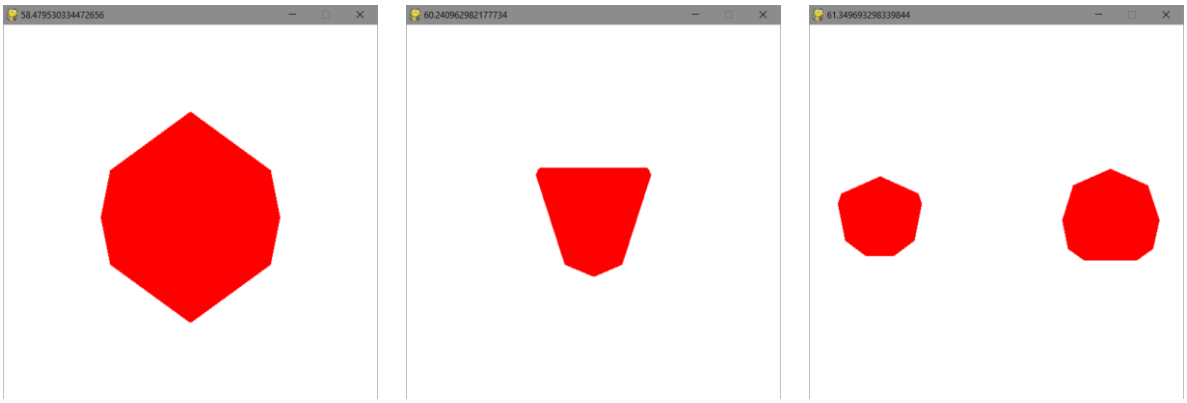
```
# A simple ThreeDMesh for testing purposes.
mesh = ThreeDMesh("1", (150,0,0), (255,0,0))
```

before the main loop, and then add

```
mesh.draw(screen)
```

in the drawing section of the main loop. Additionally, since no steering is yet implemented, we will have to set *self.z* to a non-zero value manually, in the constructor. We also need to copy one of the files generated in Exporting the levels to the level_data folder.

Here is the result after running the code with *self.z* set to a couple of different values:



self.z = 250 *self.z* = 150 *self.z* = 450

Giving the user control (2.2, 4.7)

Now I can add the update method, which will also run every refresh and update *self.z* based on the mouse y coordinate. I will start by a simple update algorithm that will just take the mouse y coordinate and directly use it as *self.y* to test the principle:

```
    # This method will update the self.y attribute based on
    # the mouse y coordinate.
    def update(self, mouse_y):
        self.z = mouse_y
```

Code 31: The basic update method for the ThreeDMesh object.

For this to work the y coordinate of the mouse's pointer will have to be passed to the method. This will be done in the main loop. I added

```
# Get the mouse coordinates.
pos = pygame.mouse.get_pos()
mouse_x = pos[0]
mouse_y = pos[1]
```

in the Event Processing section of the main loop, and then

```
# Update the mesh.
mesh.update(mouse_y)
```

in the Game Logic section.

This mechanism works and allows for basic exploration of the mesh, but as it directly responds to the movement of the mouse the changes in the world can be very rapid if the mouse is moved very fast. This could confuse both the player and the collision detection engine. Because of that I changed the update() method to reflect the one specified in the Improving the controls section.

```
66.     # This method will update the self.y attribute based on
67.     # the mouse y coordinate.
68.     def update(self, mouse_y):
69.         # Calculate the difference between the mouse position
70.         # and the current z position.
71.         diff = mouse_y - self.z
72.         # If the difference is bigger than 50,
73.         # limit the transition speed.
74.         if abs(diff) > 50:
75.             # abs(diff)/diff is used to copy the sign of diff.
76.             diff = 50 * abs(diff) / diff
77.         # Add a fraction of the difference to self.z.
78.         self.z += diff * 0.1
79.
```

Code 32: The updated code of the update() method which makes the shape transformations smoother.

This change indeed makes the transitions much smoother and pleasing to the eye, it should also be easier on the collision engine.

Adding colour (1.4, 3.5)

As specified in the Colour scheme, the game's elements' colours should change with the changes in their *self.z* parameters. In the Design section I gave an algorithm for calculating the colours. I decided to write a procedure that will perform this calculation, given a maximum, minimum and z value. The function must return an integer (that is pygame's colour format specification), so the outcome is rounded before returning it.

```
1.  # Calculate the colour component based on the z position.
2.  def calculateColour(min, max, z):
3.      return math.floor(min + z/500 * (max-min))
```

Code 33: The colour components calculating function.

Then I added a line to the update() method that utilises the above function to calculate the currentColour:

```
79.         # Set the currentColour based on self.z.
80.         self.currentColour = (
    calculateColour(self.baseColour[0], self.maxColour[0], self.z),
    calculateColour(self.baseColour[1], self.maxColour[1], self.z),
    calculateColour(self.baseColour[2], self.maxColour[2], self.z))
81.
```

Code 34: The currentColour setting part of the update() function.

The mechanism works, as presented on the screenshots below:

Figure 48: The colour updating code in action.

When I have more parts of the game working I may update the colours specified in the Colour scheme for the changes to be more visible. That will depend on how the various elements look next to each other and the feedback from my example stakeholder.

This concludes the rendering stage.

The collision system

Now that I can render the level I need to concentrate on detecting collisions. The system will be slightly different for the ground and lava surfaces, but the basics are pretty much the same. But first I need to implement a very basic *Player* class for testing purposes – if I want to detect collisions, I need two polygons: the obstacle and the player.

Basic Player class

As specified in the Testing section, we first only need a Player sprite that will follow the mouse. This way I can place it anywhere on the screen and check whether a collision is detected. For the time being I will also stop the updating of the *ThreeDMesh* so that when I move the Player around with my mouse the geometry of the level will not be changing. This can be achieved by manually setting *self.i* of *ThreeDMesh* and commenting out this line:

```
1.  # Update the mesh.
2.  # mesh.update(mouse_y)
```

The simplified *Player* class must have x and y attributes, which represent coordinates, width and height, and a list of vertices that will be used for collisions.

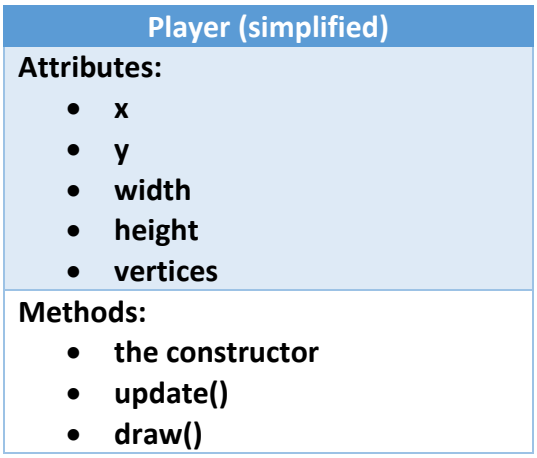


Figure 49: Class diagram for the simplified Player class.

The vertices list will be updated every refresh. This is the code for the class in Python:

```
1.  # The simplified Player class for testing purposes.
2.  class Player():
3.      def __init__(self, x, y, width, height):
4.          # Set the attributes to the values given.
5.          self.x = x
6.          self.y = y
7.          self.width = width
8.          self.height = height
9.          # Create the vertices list
10.         self.vertices = []
11.
12.     # Update the position every refresh based on mouse position.
13.     def update(self, mouse_x, mouse_y):
14.         # Set the coordinates to the mouse coordinates.
15.         self.x = mouse_x
16.         self.y = mouse_y
17.         # Calculate the coordinates of the rectangle's vertices.
18.         self.vertices = [[self.x, self.y], [self.x + self.width, self.y],
    [self.x + self.width, self.y + self.height], [self.x, self.y + self.height]]
19.
20.     # Draw the Player.
21.     def draw(self, screen):
22.        # Use pygame's built in draw rectangle function.
23.        pygame.draw.rect(screen, (0, 0, 0),[self.x, self.y, self.width, self.height]
24.
```

Code 35: The code for the simplified Player class.

Now I just need to instantiate the class before the main loop:

```
1.  player = Player(0,0,20,20)
```

Then call update() every game refresh in the Game Logic section of the main loop:

```
1.  # Game logic:
2.  # Update the mesh.
3.  # mesh.update(mouse_y)
4.  player.update(mouse_x,mouse_y)
```

And finally draw() in the drawing section of the main loop:

```
1.  # Do the drawing:
```

```
2.  screen.fill((255, 255, 255))
3.  mesh.draw(screen)
4.  player.draw(screen)
```

The class can be easily verified to work by moving the mouse cursor around – the black square representing the player instantly follows

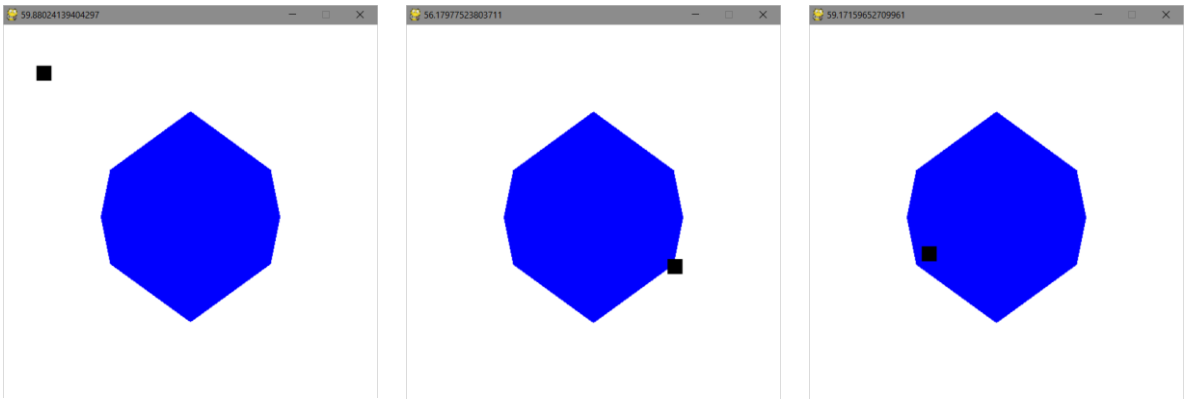


Figure 50: Moving the player's sprite around with my mouse.

Projecting polygons onto an axis

Since this is something I will need to do rather often I decided to create a separate function to handle this. Given a list of vertices it will project them on an axis given by a direction vector (called *normal*, since it will be normal to the obstacle's edge), and then return the min and max values, marking the beginning and end of the projection.

```
1.  # Project a given polygon onto an axis.
2.  def project(polygon, normal):
3.      # Create a list of projected vertices.
4.      projected = []
5.      # We treat each vertex coordinates as a position vector
6.      # and iterate on them.
7.      for vect in polygon:
8.          # Calculate the dot product of the position vector and the axis vector.
9.          dp = vect[0] * normal[0] + vect[1] * normal[1]
10.         # Calculate the projection of the position vector on the axis.
11.         projected_v = [normal[0] * dp, normal[1] * dp]
12.         # Calculate the projection's length - this is what we actually need.
13.         projected_l = math.sqrt(projected_v[0] ** 2 + projected_v[1] ** 2)
14.         # Get the direction of the projection relative to the axis direction.
15.         sign_p = projected_v[0] * normal[0] + projected_v[1] * normal[1]
16.         # Apply the direction to the projected length.
17.         projected_l = projected_l * (sign_p / abs(sign_p))
18.         # Append the calculated projection to the list of projected vertices.
19.         projected.append(projected_l)
20.     # After all vertices are processed, return the boundaries of the projection.
21.     return [min(projected), max(projected)]
```

Code 36: The function that returns the projection of a polygon on a given axis.

This function is the basis of my implementation of the Separating Axis Theorem, using it I can now start writing functions that actually detect collisions.

Check for overlap

The lava surfaces and the level surfaces have different needs when it comes to collision detection. For lava, I only need to detect whether there is a collision, whereas with the level surfaces I also need to calculate the projection vector that will get the player out of the collision. I will start with the first function, and then I will amend it to allow for projection vector calculations.

```
# Check whether there is overlap.
def checkOverlap(obstacle, player, normal):
    # Project the player and the obstacle onto the axis given by the normal vector.

    obstacle_p = project(obstacle, normal)
    player_p = project(player, normal)
    # Test for overlap.
    if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]):
        # If the above condition is true,
        # it means that the projections do not overlap.
        return False
    else:
        # Else, it means that there is overlap.
        return True

```

Code 37: The overlap-checking function.

This function returns True if the projections overlap and False if they do not. I will use it in the lava surface class, since I do not need projection vectors there – if there is a collision, the player dies, he does not need to be projected out of the lava.

One change I realised might be necessary will be checking whether the projection of the obstacle is long enough. During the Blender export process, some artefacts may have been written into the data, for example three points precisely in line. Situations like these might confuse the collision algorithm (generate very short projection vectors with no apparent source or create collisions where no obstacle is visible to the player). To counteract this I decided to add an additional condition: the projection's width must not be smaller than one pixel (the smallest renderable size). This will change the if statement from the above code to:

```
...
if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]) or obstacle_p[1]-
obstacle_p[0] < 1:
...
```

Getting the projection vector

The above function can be amended according to the Getting the projection vector (Design) section. The main difference is that if there is an overlap, the function return a list of values, not just True.

```
# Check for overlap and calculate projection vectors.
def calculateProjectionVectors(obstacle, player, normal):
    # Project the player and the obstacle onto the axis given by the normal vector.

    obstacle_p = project(obstacle, normal)
    player_p = project(player, normal)
    # Test for overlap.
```

```
7.      if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]) or
    obstacle_p[1]-obstacle_p[0] < 1:
8.          # If the above condition is true,
9.          # it means that the projections do not overlap.
10.         return False
11.     else:
12.         # Else, it means that there is overlap.
13.         # Calculate the values of the projection vectors.
14.         value1 = obstacle_p[0] - player_p[1]
15.         value2 = obstacle_p[1] - player_p[0]
16.         # Make them directed along the normal.
17.         vector1 = [normal[0] * value1, normal[1] * value1]
18.         vector2 = [normal[0] * value2, normal[1] * value2]
19.         # Return the necessary data.
20.         return [abs(value1), vector1, abs(value2), vector2]
21.
```

Code 38: Calculating the projection vectors if a collision is detected.

This function will be used in the level surfaces collision detection mechanism, where I want to push the player out of the obstacle. It returns two projection vectors and their lengths, which will be used to choose the most optimal one.

Calculating the normal

The axes that the algorithm needs to check are in fact normals to the edges of the obstacle edges and the x and y axes (normal to the player's edges). To get a normal to an edge all we need are the coordinates of its start and end. The rest of the calculations are outlined in Creating the axes, and here is the Python implementation of the process:

```
1.  # Calculate the normal vector for a given edge.
2.  def getNormal(a, b):
3.      # Obtain the vector representing the edge...
4.      edge = [b[0] - a[0], b[1] - a[1]]
5.      # ...and its length.
6.      length = math.sqrt(edge[0]**2 + edge[1]**2)
7.      # Turn the edge vector into a unit vector.
8.      edge = [edge[0] / length, edge[1] / length]
9.      # Create a vector perpendicular to the unit edge vector...
10.     normal = [-edge[1], edge[0]]
11.     # ...and return it.
12.     return normal
```

Code 39: The function that returns a unit vector normal to a given face.

The SAT library

The functions outlined above are the core of my Separating Axis Theorem implementation. They will be used in the *Level* and *Lava* classes to handle vector mathematics and collision detection. Because they are pretty much a separate part of the game I thought it would be fitting to create a library out of them.

Libraries are a way of further modularization of code without cluttering the main Python file (main.py in my case). They can be used by adding an import statement at the beginning of the code and then referring to their functions by using the syntax *name_of_library.name_of_function()*. In fact, pygame is a Python library that I am actively using already. Introducing an SAT library will make code maintenance easier, and the whole

program easier to read. It will also increase my code's modularity, as the SAT library will be available to use in other projects freely.

Creating libraries in Python is rather simple, I just need to copy all the functions to a file named *name_of_library.py* and place it in the same directory as *main.py*. Then I just need to add

```
# Import the Separating Axis Theorem library
import sat
```

at the beginning of *main.py*. The contents of *sat.py* are now:

```
# The Separating Axis Theorem library by Jakub Dranczewski.
# Developed for the Dimension Surfer game, as part of a
# Computer Science Project.

import math

# Project a given polygon onto an axis.
def project(polygon, normal):
    # Create a list of projected vertices.
    projected = []
    # We treat each vertex coordinates as a position vector
    # and iterate on them.
    for vect in polygon:
        # Calculate the dot product of the position vector and the axis vector.
        dp = vect[0] * normal[0] + vect[1] * normal[1]
        # Calculate the projection of the position vector on the axis.
        projected_v = [normal[0] * dp, normal[1] * dp]
        # Calculate the projection's length - this is what we actually need.
        projected_l = math.sqrt(projected_v[0] ** 2 + projected_v[1] ** 2)
        # Get the direction of the projection relative to the axis direction.
        sign_p = projected_v[0] * normal[0] + projected_v[1] * normal[1]
        # Apply the direction to the projected length.
        projected_l = projected_l * (sign_p / abs(sign_p))
        # Append the calculated projection to the list of projected vertices.
        projected.append(projected_l)
    # After all vertices are processed, return the boundaries of the projection.
    return [min(projected), max(projected)]

# Check whether there is overlap.
def checkOverlap(obstacle, player, normal):
    # Project the player and the obstacle onto the axis given by the normal vector.

    obstacle_p = project(obstacle, normal)
    player_p = project(player, normal)
    # Test for overlap.
    if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]) or obstacle_p[
1]-obstacle_p[0] < 1:
        # If the above condition is true,
        # it means that the projections do not overlap.
        return False
    else:
        # Else, it means that there is overlap.
        return True

# Check for overlap and calculate projection vectors.
def calculateProjectionVectors(obstacle, player, normal):
    # Project the player and the obstacle onto the axis given by the normal vector.

    obstacle_p = project(obstacle, normal)
    player_p = project(player, normal)
```

```
48.     # Test for overlap.
49.     if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]) or obstacle_p[
    1]-obstacle_p[0] < 1:
50.         # If the above condition is true,
51.         # it means that the projections do not overlap.
52.         return False
53.     else:
54.         # Else, it means that there is overlap.
55.         # Calculate the values of the projection vectors.
56.         value1 = obstacle_p[0] - player_p[1]
57.         value2 = obstacle_p[1] - player_p[0]
58.         # Make them directed along the normal.
59.         vector1 = [normal[0] * value1, normal[1] * value1]
60.         vector2 = [normal[0] * value2, normal[1] * value2]
61.         # Return the necessary data.
62.         return [abs(value1), vector1, abs(value2), vector2]
63.
64. # Calculate the normal vector for a given edge.
65. def getNormal(a, b):
66.     # Obtain the vector representing the edge...
67.     edge = [b[0] - a[0], b[1] - a[1]]
68.     # ...and its length.
69.     length = math.sqrt(edge[0]**2 + edge[1]**2)
70.     # Turn the edge vector into a unit vector.
71.     edge = [edge[0] / length, edge[1] / length]
72.     # Create a vector perpendicular to the unit edge vector...
73.     normal = [-edge[1], edge[0]]
74.     # ...and return it.
75.     return normal
76.
```

Code 40: The Separating Axis Theorem library.

To use the functions in this library I must now for example write:

```
1.  normal = sat.getNormal([200,300], [500,120])
```

Using the collision system

I can now use the *sat* library to implement the collision for the lava surfaces and level surfaces, which will both inherit from the *ThreeDMesh* class.

The lava surfaces (1.6, 4.5)

I will now collide a *collide()* method, that will be an additional method in the *Lava* class, other than the methods inherited from *ThreeDMesh*. The *collide()* method will iterate on the polygons in the current cross section, so it is slightly different than the functional approach specified in Putting it all together (The Separating Axis Theorem section). I decided that since I need two slightly different collision handling approaches I should implement them in the classes themselves, not as an additional function.

The loop in the *collide()* method should check all the polygons in the current cross-section until it finds a collision. When a collision is found, the loop will be stopped and the optimal projection vector returned. If it is found that for a certain polygon there is no overlap on a given axis, we can jump to the next one. Instead of the *return [0,0]* statement used in the pseudocode we can use Python's *continue* for that (it breaks the current iteration and jumps to the next one). This can be summarized in a flowchart:

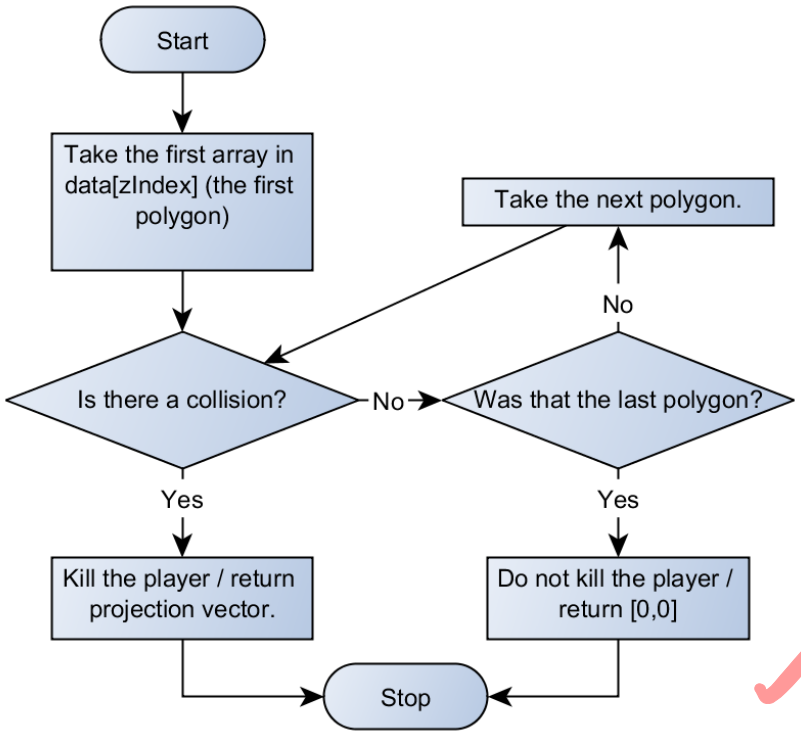


Figure 51: The collision checking iteration process.

For testing purposes I decided to simply turn the colour of the polygon to red if there is a collision. This is the Python code of the *Lava* class:

```
1.  # The class for the lava surfaces.
2.  class Lava(ThreeDMesh):
3.      # A method for detecting collisions.
4.      def collide(self, player):
5.          # Set colour to blue for debugging purposes
6.          self.currentColour = (0, 255, 0)
7.          # Take the current cross-section from the data array.
8.          cSection = self.data[math.floor(self.z)]
9.          # Iterate over the polygons in the current cross-section.
10.         for obstacle in cSection:
11.             # Check the x and y axes.
12.             if not sat.checkOverlap(obstacle, player.vertices, [1,0]):
13.                 # If there is no overlap we can jump to the next
14.                 # polygon in the data set thanks to the SAT principles.
15.                 continue
16.             if not sat.checkOverlap(obstacle, player.vertices, [0,1]):
17.                 continue
18.             # Iterate over the polygon's edges.
19.             # We assume that there is overlap unless proven otherwise.
20.             collided = 1
21.             for i in range(len(obstacle)):
22.                 # Get the normal to this edge...
23.                 normal = sat.getNormal(obstacle[i], obstacle[(i+1) % len(obstacle)])

24.                 # ...and check for overlap, if the axis is not the x or y axis.
25.                 if not sat.checkOverlap(obstacle, player.vertices, normal):
26.                     # Stop checking the edges and rise the flag that
27.                     # there is no overlap.
28.                     collided = 0
29.                     break
30.             # If we got past all the overlap checks and there was overlap
31.             # on all the axes, it means that there is a collision, so we
```

```
32.             # turn the polygon red (for testing).
33.             if collided:
34.                 self.currentColour = (255, 0, 0)
35.                 # If there is a collision we do not need to check
36.                 # the rest of the polygons.
37.                 break
38.
```

Code 41: The collision checking algorithm in the Lava class.

To test the above code I need to make some small changes to the rest of the code:

```
1.  # A simple Lava object for testing purposes.
2.  mesh = Lava("1", (0,0,255), (0,255,0))
```

After moving the mouse a little bit the game crashed. The error printed by Python stated:

```
  File "D:\Kuba\aplikacje\GitHub\Dimension-Surfer-Project\sat.py", line 33, in checkOverlap
    obstacle_p = project(obstacle, normal)
  File "D:\Kuba\aplikacje\GitHub\Dimension-Surfer-Project\sat.py", line 24, in project
    projected_l = projected_l * (sign_p / abs(sign_p))
ZeroDivisionError: float division by zero

Process finished with exit code 1
```

It turns out that if the projection of a point has value 0, the dot product of the projection and normal also goes to zero, and since I divide by it to get the correct sign for the projection, this results in an error. To solve this I decided to use Python's *copysign()* function, which copies the sign of one variable to another, accounting for zeros. The faulty line in *project()* will thus now be:

```
23. ...
24. projected_l = math.copysign(projected_l, sign_p)
25. ...
```

Another improvement I realised can be made is not checking the x and y axis again – if the obstacle has a horizontal or vertical edge, x or y axis will be checked again, wasting power. To prevent that a simple check can be added to the if statement in line 25 of the Lava class:

```
23. ...
24. # ...and check for overlap, if the axis is not the x or y axis.
25. if (normal[0]*normal[1] != 0) and not sat.checkOverlap(obstacle, player.vertices, no
    rmal):
26. ...
```

Thanks to short-circuit evaluation, if the first statement will be false (the normal is horizontal or vertical), the second one will not be even executed, saving computing power.

After these changes the Lava class collision system works as expected:

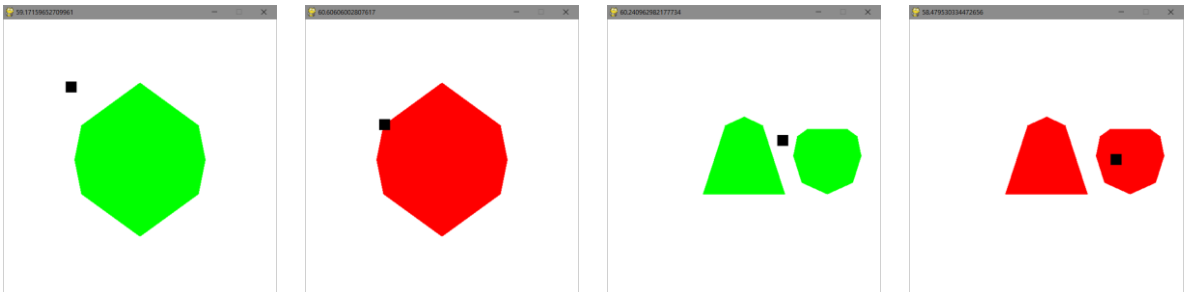


Figure 52: The shapes get red when the player overlaps them and green when there is no overlap.

Later in the development I will change the *Lava* class code so that instead of changing its colour it resets the whole level.

The level surfaces (4.1)

The process for the level surfaces is slightly more complicated since it involves projection vectors, but I will build on what I have already done with the *Lava* class. The *Level* class will be very similar, but it will store the projection vectors and in the end, call a method of the *Player* class to displace the sprite. For the time being I will create a placeholder method for that.

```
class Player():
...
    # Displace the player after collision.
    def collisionDisplace(self, projectionVector):
        # For debugging purposes print the collision vector.
        print(projectionVector)

```

Now for the *Level* class:

```
# The class for the level surfaces.
class Level(ThreeDMesh):
    # A method for detecting collisions.
    def collide(self, player):
        # Take the current cross-section from the data array.
        cSection = self.data[math.floor(self.z)]
        # Iterate over the polygons in the current cross-section.
        for obstacle in cSection:
            # Create lists for holding projection vector lengths
            # and the vectors.
            projectionVectorsLenghts = []
            projectionVectors = []
            # Check the x and y axes.
            vectors =
sat.calculateProjectionVectors(obstacle, player.vertices, [1, 0])
            # If the calculateProjectionVectors function did not return false,
            # it means that it successfully found projection vectors...
            if vectors:
                # ...which we can add to our lists.
                projectionVectorsLenghts.append(vectors[0])
                projectionVectors.append(vectors[1])
                projectionVectorsLenghts.append(vectors[2])
                projectionVectors.append(vectors[3])
            else:
                continue

```

```
            vectors =
    sat.calculateProjectionVectors(obstacle, player.vertices, [0, 1])
            if vectors:
                projectionVectorsLenghts.append(vectors[0])
                projectionVectors.append(vectors[1])
                projectionVectorsLenghts.append(vectors[2])
                projectionVectors.append(vectors[3])
            else:
                continue

            # Iterate over the polygon's edges.
            # We assume that there is overlap unless proven otherwise.
            collided = 1
            for i in range(len(obstacle)):
                # Get the normal to this edge...
                normal =
    sat.getNormal(obstacle[i], obstacle[(i + 1) % len(obstacle)])
                # ...and check for overlap, if the axis is not the x or y axis.
                if (normal[0] * normal[1] != 0):
                    vectors =
    sat.calculateProjectionVectors(obstacle, player.vertices, normal)
                    if vectors:
                        projectionVectorsLenghts.append(vectors[0])
                        projectionVectors.append(vectors[1])
                        projectionVectorsLenghts.append(vectors[2])
                        projectionVectors.append(vectors[3])
                    else:
                        # Stop checking the edges and rise the flag that
                        # there is no overlap.
                        collided = 0
                        break
            # If we got past all the overlap checks and there was overlap
            # on all the axes, it means that there is a collision.
            if collided:
                # Find the index of the shortest vector...
                minimumIndex =
    projectionVectorsLenghts.index(min(projectionVectorsLenghts))
                # ...and pass it to the player.
                player.collisionDisplace(projectionVectors[minimumIndex])
                # If there is a collision we do not need to check
                # the rest of the polygons.
                break

```

Code 42: The Level class collide() method.

I also changed the *mesh* object instantiation line:

```
# A simple Level object for testing purposes.
mesh = Level("1", (0,0,255), (0,255,0))
```

The above algorithm seems to print correct projection vectors for different player positions:

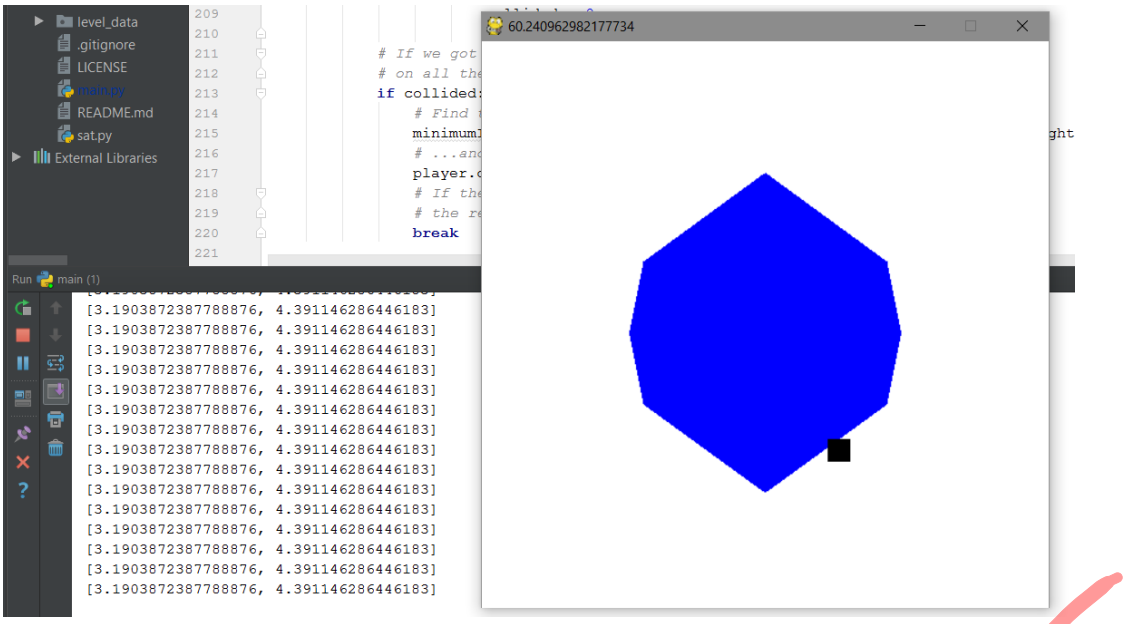


Figure 53: The output from the Level collision detection algorithm.

To test the algorithm further I decided to change the *Player*'s *collisionDisplace()* method to actually displace the sprite:

```
# Displace the player after collision.
def collisionDisplace(self, projectionVector):
    # Change the player's x and y coordinates according to the projection vector.
    self.x += projectionVector[0]
    self.y += projectionVector[1]
```

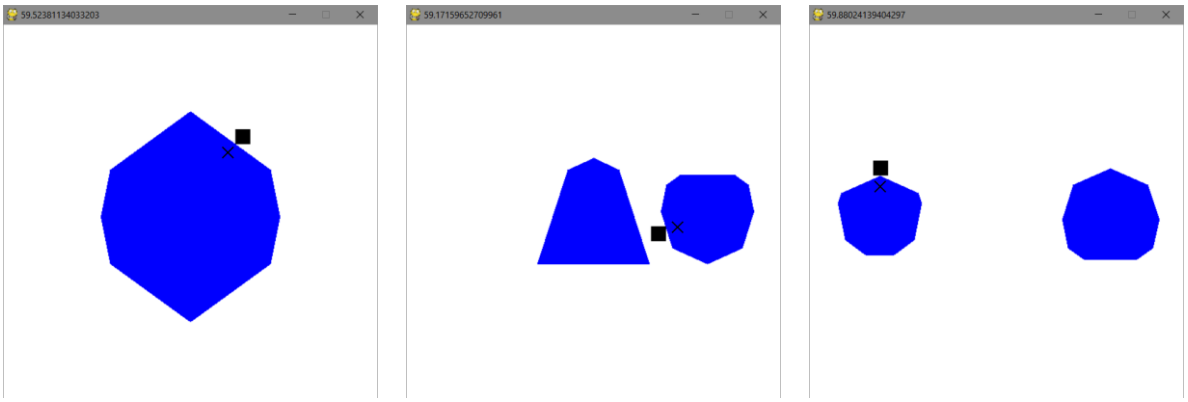


Figure 54: The displaced player sprite when it originally was in places indicated with the crosses. The process works for single and multiple polygons.

Lava and level for player implementation

Now that the *Lava* and *Level* classes are mostly implemented, I can move on to implementing a proper *Player* class. But first, I will construct a testing environment using the first level design from Design mock-up and other sections.

I have already created and exported this level in Blender, now I will create objects (before the start of the main loop) that will allow me render it:

```
# Creating objects for testing:
```

```
level = Level("1_level", (33,150,243), (13,71,161))
lava = Lava("1_lava", (255,9,9), (180,0,0))
player = Player(0,0,20,20)
```

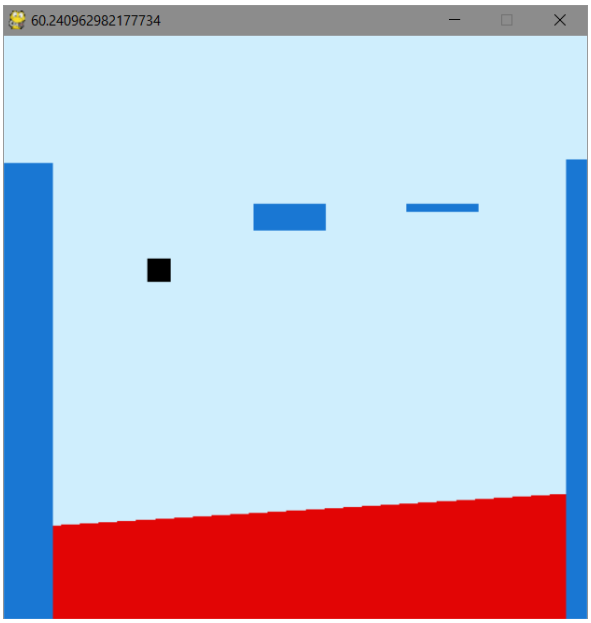
The main loop also must be updated:

```
# Game logic:
# Update level and lava based on mouse position
level.update(mouse_y)
lava.update(mouse_y)
# Move the player
player.update(mouse_x, mouse_y)
# Collide the player with the lava and the level
lava.collide(player)
level.collide(player)

# Do the drawing:
# Set the backgorund color
backgroundBaseColour = (225,245,254)
backgroundMaxColour = (179,229,252)
screen.fill((
    calculateColour(backgroundBaseColour[0], backgroundMaxColour[0], level.z),
    calculateColour(backgroundBaseColour[1], backgroundMaxColour[1], level.z),
    calculateColour(backgroundBaseColour[2], backgroundMaxColour[2], level.z)))
# Draw the lava, the level and the player
lava.draw(screen)
level.draw(screen)
player.draw(screen)

# Update the screen:
pygame.display.flip()

```

Note that I added lines 13-18 that change the colour of the background according to the Colour scheme. There is also a cosmetic change that must be done to the *Lava* class – I no longer want it to set *currentColour* to green when the *collide()* method is called, it is already reset in the *update()* method, so I can remove

```
# Set colour to green for debugging purposes
self.currentColour = (0, 255, 0)
```

Now the rendered level looks like this:



A full code listing is provided in Appendix C

Figure 55: The rendered level used for testing implementation of player's navigation.

The Player class

I have already created a simple *Player* class before for testing purposes. Now I will amend this class, adding functionality so that it is a fully fledged 'character' in the game.

The new class diagram is slightly different:

Player
Attributes: • **x** • **y** • **xSpeed** • **ySpeed** • **xAcceleration** • **yAcceleration** • **width** • **height** • **baseColour** • **maxColour** • **vertices**
Methods: • **the constructor** • **update()** • **draw()** • **collisionDisplace()** • **reset()**

continual iterative development

Figure 56: The new Player class diagram.

The additions are some new attributes: xSpeed, ySpeed, xAcceleration and yAcceleration, which will be used for navigation, and colours, but also two new methods: collisionDisplace()

(which I discussed earlier) and reset() (which will reset the player's position after colliding with lava). Some of the existing methods will also have to be altered.

The constructor

The constructor needs to be updated so that it sets all the new attributes:

```
# The Player class.
class Player():
    def __init__(self, x, y, width, height, baseColour, maxColour):
        # Set the attributes to the values given.
        self.x = x
        self.y = y
        self.xSpeed = 0
        self.ySpeed = 0
        self.xAcceleration = 2
        self.yAcceleration = 0.2
        self.width = width
        self.height = height
        self.baseColour = baseColour
        self.maxColour = maxColour
        # Create the vertices list
        self.vertices = []

    # Other methods will go here...
```

Code 43: The constructor of the Player class.

It also takes two new arguments to set the Player's colours, so the instantiation line must be updated:

```
player = Player(0, 0, 20, 20, (255,193,0), (255,111,0))
```

The colours passed are in accordance with the Colour scheme.

Drawing the player (1.3)

The method for rendering the player's sprite does not change much, but I will add the colour changes. Since the *Player* class does not hold the *self.z* attribute, I will use the *Level*'s attribute for the calculations. It means that it must be passed while the *draw()* method is called:

```
player.draw(screen, level.z)
```

And the method itself will be:

```
# Draw the Player.
def draw(self, screen, levelZ):
    # Calculate the colour to be used while drawing.
    colour = (calculateColour(self.baseColour[0], self.maxColour[0], levelZ),
              calculateColour(self.baseColour[1], self.maxColour[1], levelZ),
              calculateColour(self.baseColour[2], self.maxColour[2], levelZ))
    # Use pygame's built in draw rectangle function.
    pygame.draw.rect(screen, colour, [self.x, self.y, self.width, self.height])
```

Code 44: The method for drawing the Player.

Resetting (1.6)

When the player hits a lava surface, his position and ySpeed should be reset. This will be achieved with a *reset()* method:

```
1.  # Reset the Player's position.
2.  def reset(self):
3.      # Set the x and y coordinates to zero.
4.      self.x = 0
5.      self.y = 0
6.      # Reset the ySpeed.
7.      self.ySpeed = 0
```

Code 45: The Player's reset method.

I then need to call this method when an object of the *Lava* class detects a collision. I can do this by changing the colour-changing line in the *collide()* method of this class:

```
29. ...
30. # If we got past all the overlap checks and there was overlap
31. # on all the axes, it means that there is a collision, so we
32. # reset the level.
33. if collided:
34.     player.reset()
35.     # If there is a collision we do not need to check
36.     # the rest of the polygons.
37.     break
```

continual iterative development through prototype testing

I can now verify that this works by placing the sprite in the lava. It is instantly transported to the top left corner of the screen:

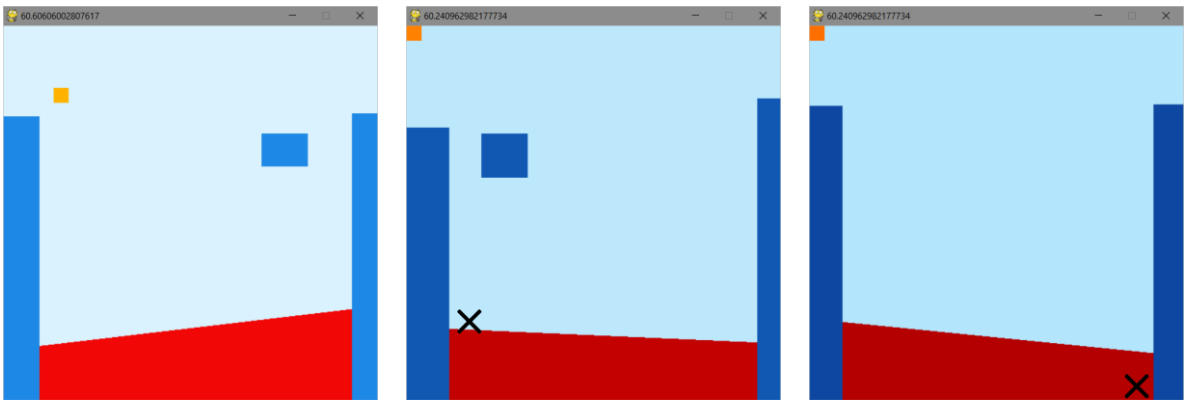


Figure 57: Testing the reset() method – the black cross shows the place where the sprite would be if not for the resetting.

Displacement after collision (4.1)

This has already been implemented earlier, in The level surfaces section (Using the collision system), and does not have to be further altered.

The navigation (2.1)

This has been discussed in a lot of detail in The navigation algorithm section. The algorithm is quite complex, but should be rewarding and easy to use for the player. I decided to split it into two separate problems: reading the inputs and updating the player object.

Reading the inputs (2.1)

I will be using the pygame input loop for this. A very simple version of it has already been specified in The core pygame code section – it is the part which checks whether the user wants to close the window. Now I will rewrite it according to The navigation algorithm and Jumping sections.

```
# Event processing - we iterate on the events given to us by pygame:
for event in pygame.event.get():
    # If the event type is QUIT, the user wants to close the window.
    # So we set done to True.
    if event.type == pygame.QUIT:
        done = True
    # Handle the keydown events.
    elif event.type == pygame.KEYDOWN:
        # Go left.
        if event.key == pygame.K_a or event.key == pygame.K_LEFT:
            leftPressed = 1
            xSpeed = -1
        # Go right.
        elif event.key == pygame.K_d or event.key == pygame.K_RIGHT:
            rightPressed = 1
            xSpeed = 1
            # Jump.
        elif event.key == pygame.K_w or event.key == pygame.K_SPACE or
                                    event.key == pygame.K_UP:
            ySpeed = -1
    # Handle the keyup events.
    elif event.type == pygame.KEYUP:
        if event.key == pygame.K_a or event.key == pygame.K_LEFT:
            leftPressed = 0
            xSpeed = 0
            # If right is still pressed, start going right.
            if rightPressed == 1:
                xSpeed = 1
        elif event.key == pygame.K_d or event.key == pygame.K_RIGHT:
            rightPressed = 0
            xSpeed = 0
            # If left is still pressed, start going left.
            if leftPressed == 1:
                xSpeed = -1
        elif event.key == pygame.K_w or event.key == pygame.K_SPACE or
                                      event.key == pygame.K_UP:
            ySpeed = 0
```

Code 46: Event processing for player navigation

leftPressed, rightPressed, xSpeed and *ySpeed* all need to be initiated since they will be used even if no button is pressed or if the other (left/right) button was not pressed before. I will thus add those lines of code before the main loop:

```
xSpeed = 0
ySpeed = 0
leftPressed = 0
rightPressed = 0
```

Now I just need to pass the speed to *Player's update()* method:

```
# Move the player
player.update(xSpeed, ySpeed)
```

and I can start implementing the movement.

Moving the player (2.1, 4.9)

This has been described in the Design section, throughout Falling, Jumping and The navigation algorithm. The general ideas are fairly straightforward, there is a problem though – the *Level collide()* method does not return the y component of the projection vector, which I want to use as a jumping condition. It does pass it to the *collisionDisplace()* though, so I can use that. *collisionDisplace()* is technically called after *update()*, so I will be using the projection vector from the previous loop, but it should not be a problem. That also means that I have to initialize the attribute in the constructor, since at the first run of *update()* it will not exist otherwise. I added this line to the constructor:

```
1.  self.yPV = 0
```

and changed the *collisionDisplace()* method:

```
1.  # Displace the player after collision.
2.  def collisionDisplace(self, projectionVector):
3.      # Change the player's x and y coordinates according to the projection vector.
4.      self.x += projectionVector[0]
5.      self.y += projectionVector[1]
6.      # Save the y component of the projection vector for use in update()
7.      self.yPV = projectionVector[1]
```

This will have to be reset every *update()* since *collisionDisplace()* is not called every frame. The method is now:

```
1.  # Update the position every refresh based on keyboard input.
2.  def update(self, xSpeed, ySpeed):
3.      # Set xSpeed based on left/right keys pressed.
4.      self.xSpeed = xSpeed * self.xAcceleration
5.      # Add a constant to the ySpeed to simulate freefall.
6.      self.ySpeed += self.yAcceleration
7.      # Jump if conditions are met.
8.      if ySpeed == -1 and self.yPV < 0:
9.          self.ySpeed = -5
10.     # Reset the stored y component of the projection vector.
11.     self.yPV = 0
12.     # Add the speeds to the coordinates.
13.     self.x += self.xSpeed
14.     self.y += self.ySpeed
15.     # Calculate the coordinates of the rectangle's vertices.
16.     self.vertices = [[self.x, self.y], [self.x + self.width, self.y],
    [self.x + self.width, self.y + self.height], [self.x, self.y + self.height]]
```

I run this and tried going in a few directions. An obvious problem surfaced instantly – there are no limits on the ySpeed and after a few seconds the player starts moving rather randomly, simply flying through obstacles. Two changes should solve that: introducing a maximum speed and resetting the speed every collision. I added this condition to the *update()* method:

```
6.  ...
7.  # Set a speed limit.
8.  if self.ySpeed > 5:
9.      self.ySpeed = 5
10. ...
```

This solved the uncontrollable speed problem. Now I will add resetting the speed on collision and the system should work perfectly. These are the lines I added to the *collisionDisplace()* method:

```
# Reset the ySpeed after collision
self.ySpeed = 0
```

This proved to make the movement around the corners way smoother (the player sometimes just fell before reaching the end of an obstacle, this is solved now), but introduced a new challenge – the player sticks to walls. Since the ySpeed is reset after each collision, it is also reset in this situation:

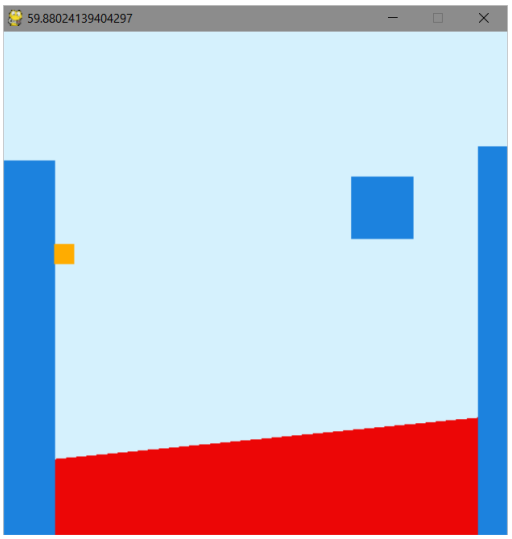


Figure 58: Player sticking to a wall.

To solve that and at the same time make the falling speed more intuitive on steep slopes (instead of just making it incredibly slow). To do that I resorted to trigonometry:

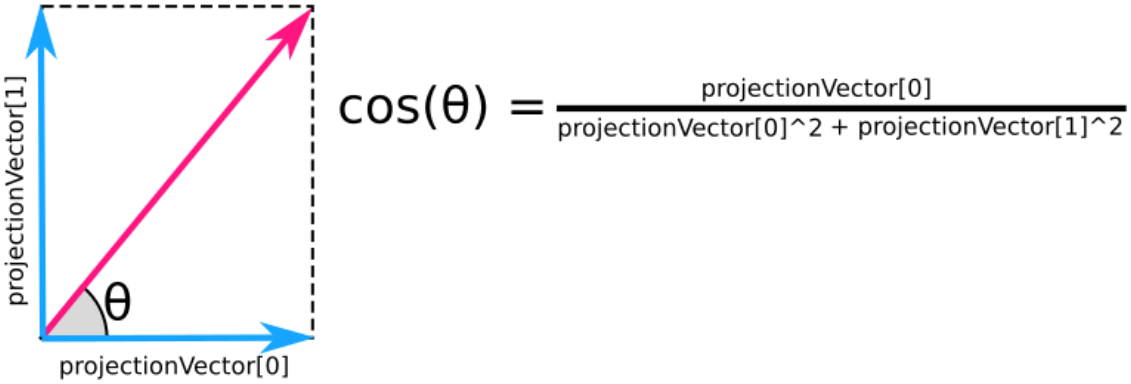


Figure 59: The cosine function.

The cosine function of the indicated angle will be zero for a surface that is perfectly horizontal and one if the surface is perfectly vertical. By multiplying *ySpeed* by this value I should achieve natural looking speed reduction for surfaces of different inclinations.

I changed the above speed resetting code in *collisionDisplace()* to:

```
1.  if projectionVector[1] < 0:
2.      self.ySpeed *= abs(projectionVector[0]) / math.sqrt(projectionVector[0] ** 2 + p
    rojectionVector[1] ** 2)
```

This also checks whether the y component of the collision vector is smaller than zero (in pygame the y axis is directed downwards) to prevent division by zero. The absolute value of the x component is taken since we only need a percentage in the end, direction is not necessary.

It is hard to demonstrate the workings of the new method on paper, but I tested it on various surfaces, including vertical, horizontal and inclined in different ways, and it behaves as desired.

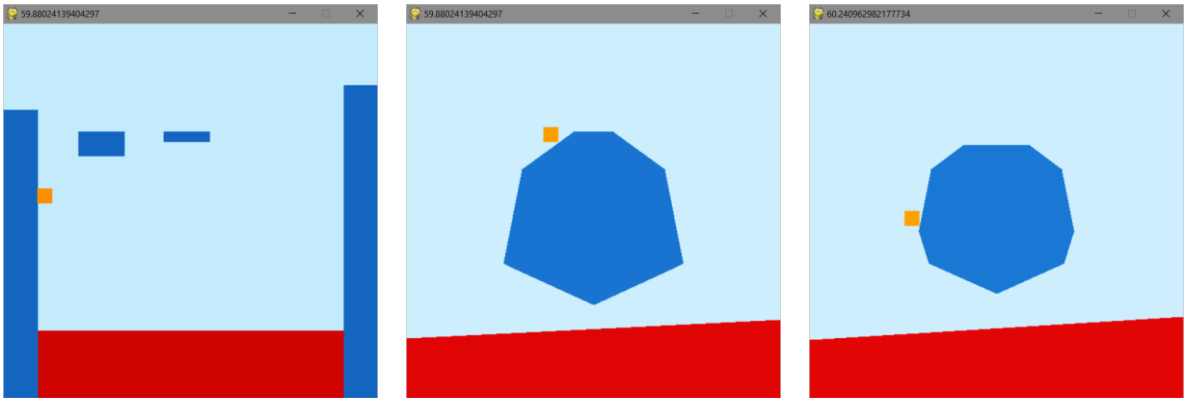


Figure 60: Testing the new speed reducing equation on different surfaces.

Testing this solution brought to my attention another problem – the jumping condition. It simply checks if the y component of the projection vector is non-zero and facing in the right direction. But what if the slope is very steep and the y component is incredibly small, but still non-zero? The wall may look vertical, but the player might still be able to jump off it given that it is even very slightly inclined. As a simple fix I changed the critical y component value and this proved to work just fine – it prevented me from being occasionally able to jump while falling nearby the side walls of the test level.

```
1.  # Jump if conditions are met.
2.  if ySpeed == -1 and self.yPV < -0.1:
3.      self.ySpeed = -5
```

After all these changes the collision system seems to be completely up and running, which allows me to move on to implementing the more game-centric features.

ongoing testing and refinement throughout the development

Stars

The details of the star mechanism were outlined in Winning conditions. The player is able to collect stars and collecting them is the main goal of the game. I decided to create a *Stars* class for handling this. It will need to read the star data from a file and later display them along with a score at the top of the screen. This is the class diagram I came up with based on the needs:

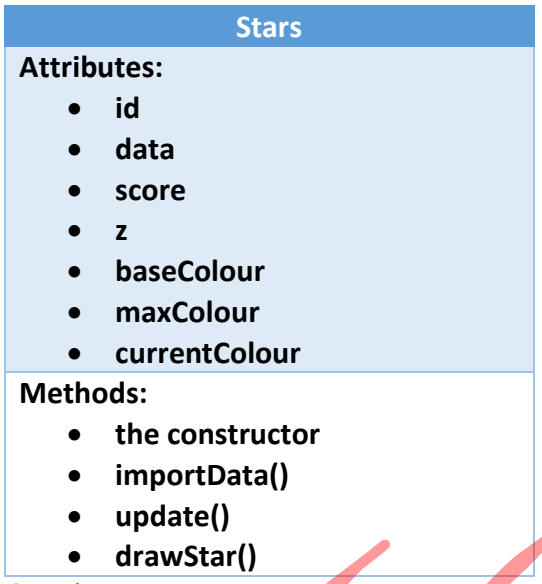


Figure 61: Class diagram of the Stars class.

The problem can again be split into several sub problems.

ongoing problem decomposition and justification of approach

The constructor

The constructor of the *Stars* class is used to set the attributes and call the *importData()* method.

```
1.  # The class that handles all things related to stars
2.  class Stars():
3.      def __init__(self, id, baseColour, maxColour):
4.          # Set the attributes
5.          self.id = id
6.          self.z = 0
7.          self.score = 0
8.          # Import the data from a text file
9.          self.data = self.importData()
10.         self.baseColour = baseColour
11.         self.maxColour = maxColour
12.         self.currentColour = self.baseColour
13.
14.     # Other methods will go here...
```

Code 47: The constructor of the Stars class.

Note that the constructor calls *importData()*, a method that will be used to populate *self.data* with the data about stars positions.

Storing the star data (4.6)

First I should establish a format for storing the stars coordinates. In Exporting the levels I already created a *1_stars.txt* file. I think it will be optimal to store the coordinates of stars in separate lines, x and y coordinates separated with spaces. For example:

```
110 150
300 400
310 120
```

These are just random numbers, but while I design the levels I would need to place the stars rather precisely. To do that without trial and error I can add a single line to the main loop:

```
1.  print(mouse_x, mouse_y)
```

I can comment out this line when I do not need it and uncomment it while placing stars. This is the result:

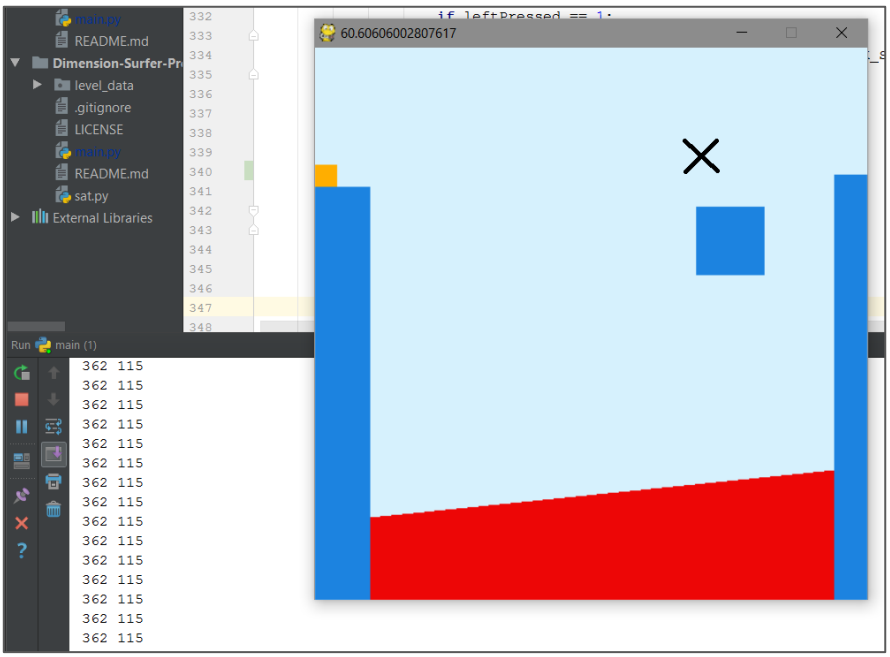


Figure 62: The coordinates of the mouse cursor are printed out.

This way I obtained the coordinates for the three stars in the first level – I want them to be above the three platforms:

```
115 106
223 110
358 109
```

Now that I have the file I can import the data.

Importing the star data (4.6)

The import process will create a list of lists containing data about particular stars – their x and y coordinates and whether they have been collected.

```
14. # Import the data about the stars
15. def importData(self):
16.     # Create a temporary data list.
17.     data = []
18.     # Open the data file.
19.     with open('level_data/' + self.id + ".txt", 'r') as f:
20.         # Parse every line in the file.
```

```
        for line in f:
            # Create a temporary list for every star and append coordinates
            star = [int(x) for x in line.split(" ")]
            # Append the state of the star. 0 is uncollected.
            star.append(0)
            # Append the star list to the data list.
            data.append(star)
    # Ensure that the file is closed.
    f.closed
    # Return the data array
    return data
```

Code 48: The data importing method in the Stars class.

The above code iterates on the lines of the stars file and stores the data in the list, which is then returned. The constructor then stores it in the *self.data* attribute.

Rendering a single star

In Rendering the star score section I wanted to use bitmaps for rendering empty and full stars. Now I see that it would prevent me from properly changing their colour. This would break the colour scheme, since the stars' colour should change in shade with the rest of the level. I will thus use vector graphics for the stars.

Pygame has a method for drawing polygons, which I already used for drawing the level and lava. I can use the same function to draw the empty and full stars. Using http://jxnblk.com/paths/ I created a star shape[7] and rewritten it to a list of vertices in the format accepted by pygame:

```
[[32,10],[20,10],[16,0],[12,10],[0,10],[9,19],[6,30],[16,24],[26,30],[23,19],[32,10
]]
```

To draw that star at a given coordinate I will have to add the x and y coordinate to the respective coordinates in the list. The *drawStar()* function is then:

```
# Draw a single star.
def drawStar(self, screen, x, y, state):
    # The basic star vertices
    vertices = [[32,10],[20,10],[16,0],[12,10],[0,10],[9,19],[6,30],[16,24],[26,30],
[23,19],[32,10]]
    # Add the given x and y coordinates to the star's vertices.
    correctedVertices = [[v[0] + x, v[1] + y] for v in vertices]
    # Draw the star. The state is used as width. If it is 0 (uncollected),
    # the polygon is filled, if it is 1 (collected), a border of width 3 is drawn.
    pygame.draw.polygon(screen, self.currentColour, correctedVertices, state*3)

```

Code 49: The method for drawing a single star.

This method can be used both for rendering the collectible stars and for rendering the score.

[7] Direct link to the path I created:
http://jxnblk.com/paths/?d=M64%2020%20L40%2020%20L32%200%20L24%2020%20L0%2020%20L18%2038%20L12%2060%20L32%2048%20L52%2060%20L46%2038%20Z

Drawing the collectible stars (1.2)

To draw the collectible stars I will iterate on the *data* array and call the *drawStar()* method. The elements of this array have the star coordinates and states, which are all we need.

```
43. # Draw the stars and the star score.
44. def draw(self, screen):
45.     # Iterate on the stars in the data array.
46.     for star in self.data:
47.         # Draw the stars.
48.         self.drawStar(screen, star[0], star[1], star[2])
```

Code 50: Drawing the stars stored in the data array.

By instantiating the stars object:

```
1.  stars = Stars("1_stars", (255,238,88), (253,216,53))
```

and then adding a draw call to the main loop:

```
1.  # Draw the lava, the level, stars and the player
2.  lava.draw(screen)
3.  level.draw(screen)
4.  stars.draw(screen)
5.  player.draw(screen, level.z)
```

and test whether rendering works. It does, as demonstrated below:

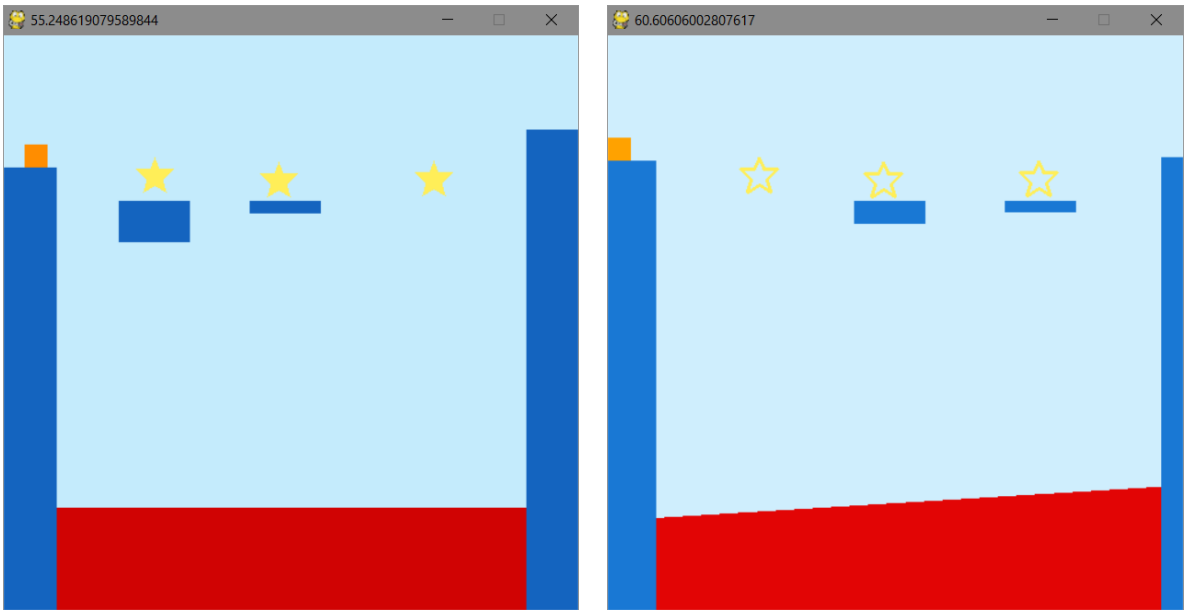


Figure 63: Rendering the stars. In the second case I manually set the collected parameter of the stars to one to check whether it will work.

Rendering the star score (3.2)

The score is stored in *self.score*, and I can use the process described in Rendering the star score to show it to the user. The amended *draw()* method will be now:

```
50. # Draw the stars and the star score.
51. def draw(self, screen):
52.     # Iterate on the stars in the data array.
```

```
    for star in self.data:
        # Draw the stars.
        self.drawStar(screen, star[0], star[1], star[2])
    # Render the star score.
    for i in range(len(self.data)):
        if i > self.score-1:
            # Draw an empty star.
            self.drawStar(screen, 5+i*40, 5, 1)
        else:
            # Draw a full star.
            self.drawStar(screen, 5+i*40, 5, 0)
```

Code 51: The complete draw() method.

This are the results:

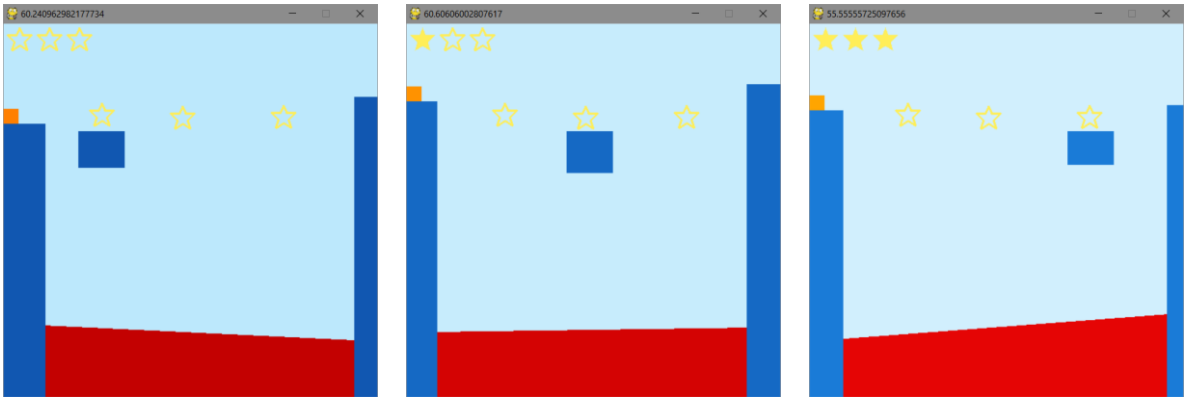


Figure 64: Rendering the star score. I set the attribute to several values manually to test it.

Collecting the stars (1.2, 4.3, 4.4)

As said in Winning conditions, the stars collision algorithm is very similar to the Separating Axis Theorem, I just need to use only the x and y axes. This also solves the projection problem, since the projections are pretty much given by player's and stars' x and y coordinates and sizes. Then the check devised in Detecting overlap can be used. I decided to create an update method for that (it will also update *self.currentColour*)

```
# Update the stars.
def update(self, mouse_y, player):
    # Use the self.z and colour updating algorithm from the ThreeDMesh class.
    diff = mouse_y - self.z
    if abs(diff) > 50:
        diff = 50 * abs(diff) / diff
    self.z += diff * 0.1
    self.currentColour = (
        calculateColour(self.baseColour[0], self.maxColour[0], self.z),
        calculateColour(self.baseColour[1], self.maxColour[1], self.z),
        calculateColour(self.baseColour[2], self.maxColour[2], self.z))
    # Iterate on the stars to check for collisions.
    for star in self.data:
        # Check for overlap. We get the projections by adding
        # width and height to respective coordinates.
        if not star[2] and not (player.x + player.width < star[0])\
                and not (player.x > star[0] + 32)\
                and not (player.y + player.height < star[1])\
                and not (player.y > star[1] + 32):
            # If there is a collision, set the star to collected...
            star[2] = 1
            # ...and add one to the score.
```

```
87.             self.score += 1
88.
```

Code 52: Updating the stars and the star score.

As noted before, I used the colour updating algorithm from *ThreeDMesh* to calculate *self.z* and *self.currentColour*, and the collision checking condition checks for overlap of 'projections', which are simply obtained by adding width to the x coordinate and height to the y coordinate when needed. The mathematical details of determining whether there is overlap are explained in Detecting overlap.

The algorithm works, I can collect stars by colliding with them, which makes them turn empty and increases the star score by one.

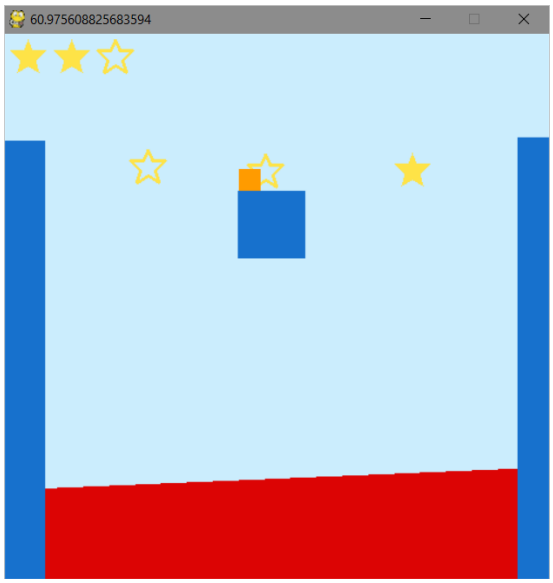


Figure 65: Collecting the stars.

Resetting after death (1.6)

As discussed in the interview, death (colliding with lava) should be punished not only by returning to the level's beginning, but also by resetting the score. This can be accomplished by creating a reset method.

```
1.  # Reset the stars' state and the star score.
2.  def reset(self):
3.      # Set the score to zero.
4.      self.score = 0
5.      # Iterate on the stars...
6.      for star in self.data:
7.          # ...setting their state to uncllected.
8.          star[2] = 0
```

Code 53: The Stars class reset method.

This now needs to be called from *Lava*'s *collide()* method. It means that the method needs a reference to the *stars* object:

```
1.  # The class for the lava surfaces.
2.  class Lava(ThreeDMesh):
```

```
3.      # A method for detecting collisions.
4.      def collide(self, player, stars):
5.  ...
6.              # If we got past all the overlap checks and there was overlap
7.              # on all the axes, it means that there is a collision, so we
8.              # reset the level.
9.              if collided:
10.                 player.reset()
11.                 stars.reset()
12. ...
```

I also need to pass the object when I call the method:

```
1.  # Collide the player with the lava and the level
2.  lava.collide(player, stars)
3.  level.collide(player)
```

All of this ensures that the star score is in fact set to zero and the stars go back to the uncollected state:

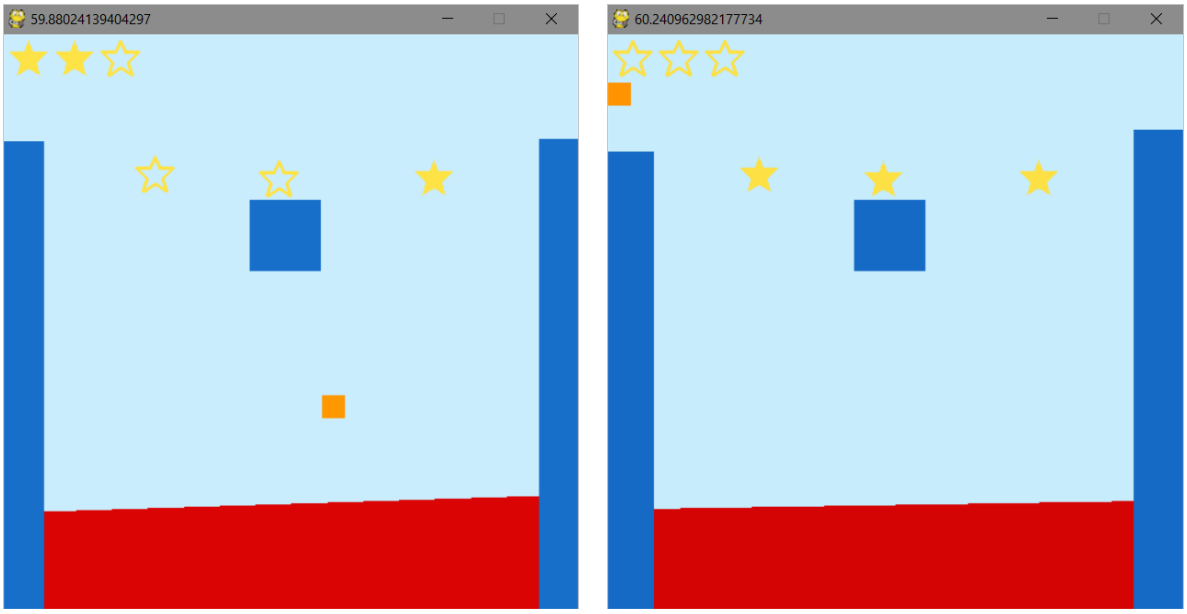


Figure 66: Star score and star states right before and right after a collision with lava.

This concludes the first stage of the implementation. Now I will consult my key user to gain some feedback on the process and ask for suggestion on improvements of the game mechanic and the first level. After that I will implement other elements of the game – the main screen and saving high scores.

iterative approach and modifications

First stage evaluation with the key user

At this stage most of the gameplay mechanics are implemented – navigation system, level manipulation, collision system, collecting stars. I decided that this is the right moment to get some feedback from my key user before I start work on the user interface and saving high scores.

I showed Krzysztof Oliwa (my key user) three test levels – the first level described above, the icosphere test level introduced in Exporting the levels and an additional test level that has

more complex geometry to test whether the more complicated navigation is intuitive for him. Here is the new test level in 3D and one of the 2D frames:

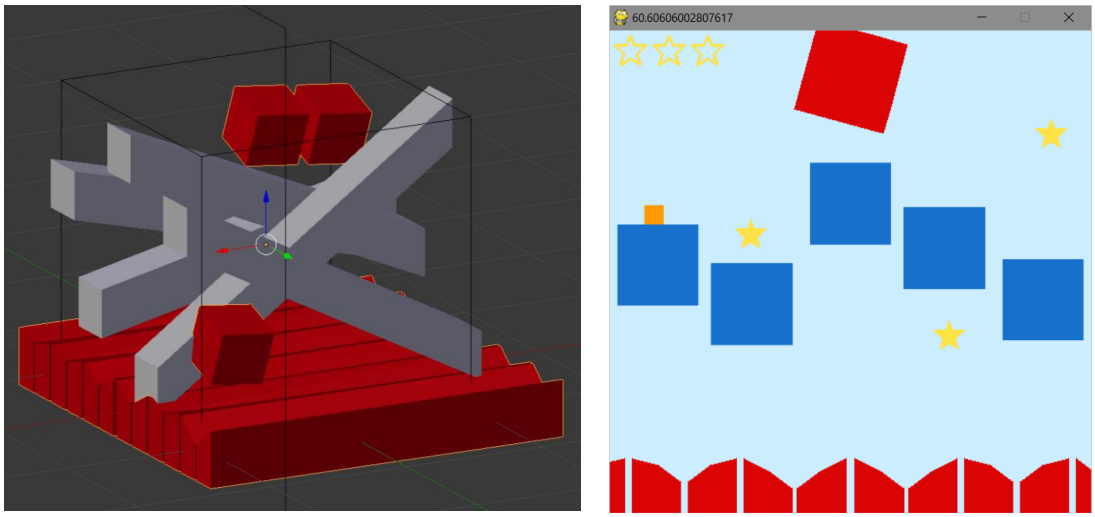


Figure 67: The test level created for the consultations.

Results of the consultations

My key user thinks that I am doing well with fulfilling the requirements. He agrees that implementing the user interface (especially the winning screen) is crucial for the game to be playable. He finds the navigation intuitive and simple, although the mouse mechanism take time to get used to (which is normal, since the concept is rather abstract). He also held the belief expressed in the interview that star score should be reset after death, saying that the levels are short enough that the loss is not too dramatic for the user.

The key user noted two problems that I have not noticed before. The first one is that the player can fall out of the screen on the left side and then there is no way to return.

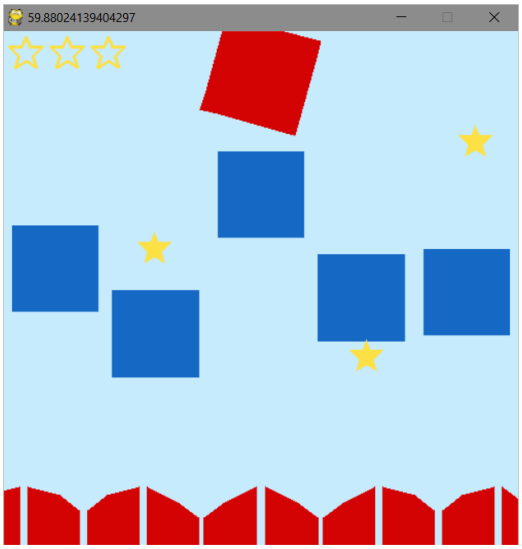


Figure 68: The test level without a player after jumping out of the screen to the left.

Thus, getting out of the screen on the left side should be disallowed.

The second problem is that the collision system is created to detect only one collision per frame. This creates problematic situation like this:

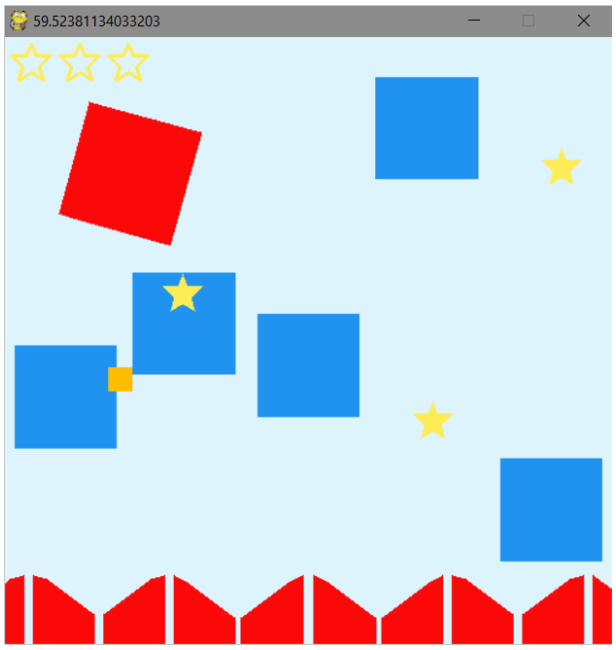


Figure 69: A collision with two objects at the same time resulting in an artefact.

This also has to be solved since it means that it is possible for the player to randomly fall through platforms.

The final suggestion of the key user was to move one of the stars in the first level slightly so that it requires a jump to obtain it. Thanks to that collecting the stars will not be completely effortless and more rewarding.

This concludes the feedback that I received from my key user. He generally enjoys the gameplay and pointed out some fixes, which I am going to implement now,

Implementing the suggestions

First, moving the star. This is very simple, since it requires me to simply change the y coordinate of one of the stars. The new *1_stars.txt* file contents are:

```
115 106
223 30
358 109
```

And the first level now looks like this:

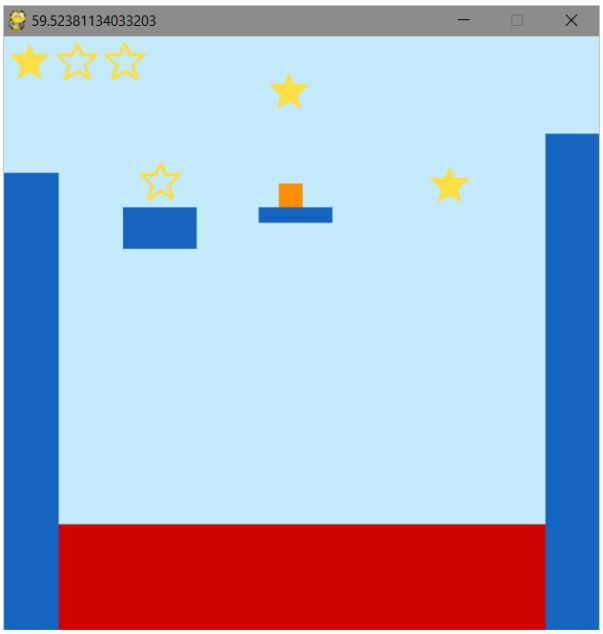


Figure 70: The first level after moving one of the stars.

Now for not allowing the user to fall out from the left side of the screen: this can be accomplished by checking whether the player's x coordinate is smaller than zero and changing it back to zero if this is true.

```
1.  ...
2.  # Add the speeds to the coordinates.
3.  self.x += self.xSpeed
4.  self.y += self.ySpeed
5.  # Check if the player is not slightly out of the screen on the left side.
6.  if self.x < 0:
7.      # Displace back onto the screen if yes.
8.      self.x = 0
9.  ...
```

Code 54: Lines 5-8 were added to the player's update() method to prevent going out of the screen on the right side.

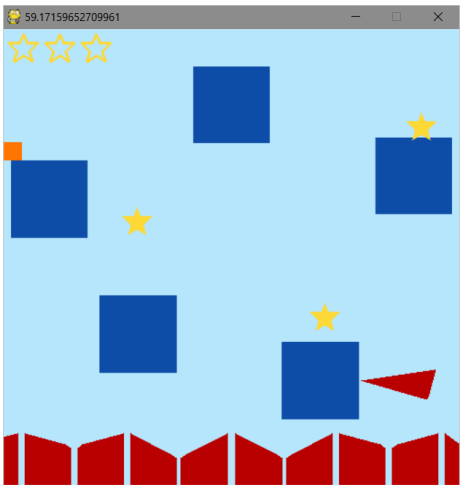


Figure 71: Pressing the left arrow key or 'a' while standing on the left border of the screen results in no movement.

Note that going out of the screen on the right side is desirable, since it is the condition for winning the game.

The last problem that I will solve is colliding with multiple polygons at the same time. This is significantly more complicated, since it requires me to modify the collision system.

Based on my knowledge of vector mathematics I decided that the best solution will be to not stop the collision detection loop after a collision is detected, but rather to collect all the projection vectors from different polygons and sum them together. This will give a projection vector that should displace the player out of all the collisions. The principle is illustrated below.

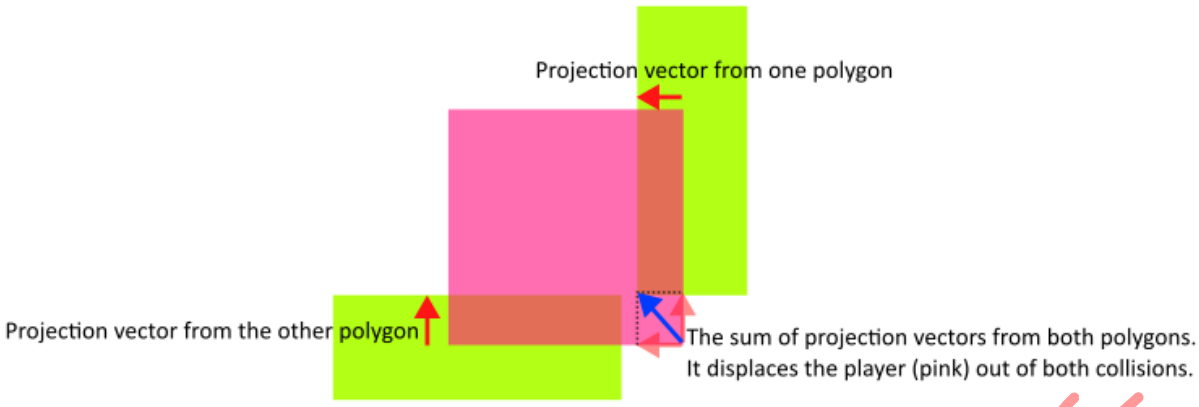


Figure 72: The principle of summing projection vectors.

To implement this I will create a *finalVector[]* list and add the projection vectors to it, and call *player.collisionDisplace()* only when all the polygons are processed, with the *finalVector* as an argument.

As a flow chart this is

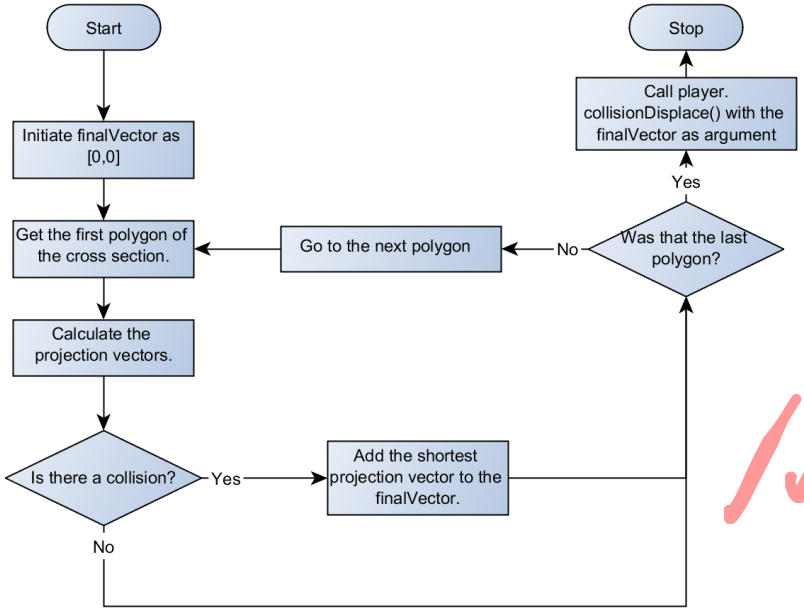


Figure 73: The new collision detection and projection vector obtaining procedure.

To implement that, two changes are needed in the *Lava* class *collide()* method. First, I need to create a *finalVector* list at the beginning of each polygon's iteration. Then I need to add the projection vector to the *finalVector* at the end of each polygon's iteration and finally call

Player's *collisionDisplace()* at the very end. The new version of the *Level* class's *collide()* method is below:

```
1.  # A method for detecting collisions.
2.  def collide(self, player):
3.      # Take the current cross-section from the data array.
4.      cSection = self.data[math.floor(self.z)]
5.      # The final projection vector will be a sum of all the projection
6.      # vectors from the collided polygons.
7.      finalVector = [0,0]
8.      # Iterate over the polygons in the current cross-section.
9.      for obstacle in cSection:
10.         ...  # This part creates projection vectors, it remains unchanged.
11.         # If we got past all the overlap checks and there was overlap
12.         # on all the axes, it means that there is a collision.
13.         if collided:
14.             # Find the index of the shortest vector...
15.             minimumIndex = projectionVectorsLenghts.index(min(projectionVectorsLengh
    ts))
16.             # ...and add it to the final projection vector.
17.             finalVector[0] += projectionVectors[minimumIndex][0]
18.             finalVector[1] += projectionVectors[minimumIndex][1]
19.     # Pass the final projection vector to the player object.
20.     # If there were no collisions, it will be [0,0], resulting in no displacement.
21.     player.collisionDisplace(finalVector)
```

Code 55: The amended Level's collide() method, now supports colliding with multiple polygons.

To test that I used the test level crated for the consultations, which has many situations in which two collisions occur at the same time. The new solution works properly.

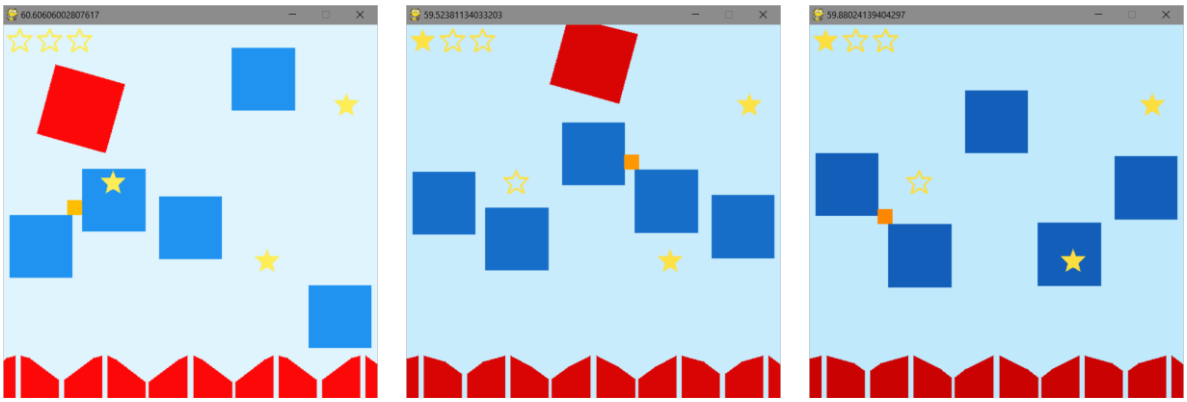


Figure 74: Testing the new collision method. The player does not fall through the blocks and cannot move through them.

Now that I have implemented the changes suggested by the key user, I will move on to developing the user interface.

modifications implemented and tested in light of user feedback

User interface

Since the game part of the program is ready and accepted by the key user, I can now proceed to creating the user interface that will help the user interact with the game.

The game's lifecycle

As explained in The game's lifecycle section in Design, a *state* variable will be introduced to store the game's state. Based on this, different actions will be undertaken in the main loop.

The advantage of this solution is that I can easily switch between states by changing the variable. The code for the skeleton of this is:

```
state = 0
# Main program loop, runs until the close button is pressed.
while not done:
    if state == -1:
        # Show the winning screen.
    elif state == 0:
        # Show the main screen.
    elif state < 9:
        # Show the level indicated by the state variable.
...
```

Code 56: The skeleton of the lifecycle management condition.

I already have most of the code for showing the level, now I have to write code that will display the main screen and seamlessly switch between states.

The high score file structure (4.8)

This has been discussed in Saving high scores and now I will create a simple *scores.txt* file for testing purposes. The contents are:

```
0 -1 -1 -1 -1 -1 -1 -1
```

Which means that the first level is unlocked and the star score is zero. These will be the initial contents of this file when the game is launched for the first time.

Before the main loop starts I will read this file and store its contents (as integers) in a list called *scores*:

```
s = open("scores.txt", 'r')
scores = [int(x) for x in s.read().split(" ")]
s.close()
```

Code 57: Reading the high score file.

Rendering the main screen (3.1)

I have already created the graphics for the main screen, as described in Design of the main screen:

Figure 75: The background and locked icon that I will use for the main screen.

This image is static, meaning that I need to render it only once – when the main screen state is entered. To do that I decided to use a variable *firstDraw*, which will be one if the screen needs to be rendered, and zero at any other time. Here is the rendering code:

```
# Load the necessary images.
backgroundImage = pygame.image.load("images/main_background.png").convert()
lockedImage = pygame.image.load("images/locked.png").convert_alpha()

state = 0
firstDraw = 1
# Main program loop, runs until the close button is pressed.
while not done:
    if state == -1:
        # Show the winning screen.
        print("win")
    elif state == 0:
        # Show the main screen.
        for event in pygame.event.get():
            # If the event type is QUIT, the user wants to close the window.
            # So we set done to True.
            if event.type == pygame.QUIT:
                done = True
        # Check if drawing needs to be done.
        if firstDraw:
            # Draw the background.
            screen.blit(backgroundImage, [0, 0])
            # Iterate on the list of scores
            for i in range(len(scores)):
                # If the level is locked display three empty stars
                # and a locked badge.
                if scores[i] < 0:
                    for j in range(3):
                        # We use the drawStar() method od the Stars class.
                        stars.drawStar(screen, 31 + i%4*113 + j*33,
285 + i//4*113, 1)
                    screen.blit(lockedImage, [28 + i%4*113, 217 + i//4*113])
                # If the level is not locked, display the star score
                # using a loop almost identical to that used in the draw()
                # method of the Stars class.
                else:
                    for j in range(3):
```

```
                        if j > scores[i] - 1:
                            stars.drawStar(screen, 31 + i%4*113 + j*33,
    285 + i//4*113, 1)
                        else:
                            stars.drawStar(screen, 31 + i%4*113 + j*33,
    285 + i//4*113, 0)
            # Refresh the screen
            pygame.display.flip()
            # Indicate that there is no need for further drawing.
            firstDraw = 0

    elif state < 9:
        # Show the level indicated by the state variable.
```

Code 58: Rendering the main screen.

I then manually altered *scores.txt* to test different cases:

Figure 76: The rendering code works, even for very random data that normally would not occur.

Processing events on the main screen (2.3)

The two kinds of useful events that may occur on the main screen are trying to close the window and a mouse click. They both can be processed using a standard pygame event processing loop. First, I will just print the number of the selected level to check whether it is calculated correctly based on the mouse position.

```
# Iterate on the events given by pygame.
for event in pygame.event.get():
    # If the event type is QUIT, the user wants to close the window.
    # So we set done to True.
    if event.type == pygame.QUIT:
        done = True
    # If the event type is MOUSEBUTTONDOWN, we assume that the user
    # may be trying to choose a level.
    elif event.type == pygame.MOUSEBUTTONDOWN:
        # Get the mouse position.
        pos = pygame.mouse.get_pos()
        mouse_x = pos[0]
        mouse_y = pos[1]
        # Check whether the cursor is in the level choice area.
        if mouse_x >= 24 and mouse_x <= 475 and mouse_y >= 217 and mouse_y <= 438:
            # Calculate the selected level inded.
            levelIndex = 4*((mouse_y-213)//113) + (mouse_x-24)//113 + 1
            # Print the level index for testing purposes.
            print(levelIndex)
```

Code 59: Handling events in the main screen state.

This is the result of running the code:

Code 60: Testing the level choice calculation. The program prints out correct numbers when clicking the levels.

Now for the actual code that will change the state to the chosen level:

```
# Calculate the selected level index.
levelIndex = 4*((mouse_y-213)//113) + (mouse_x-24)//113 + 1
# If the level is unlocked, change the state.
if scores[levelIndex-1] >= 0:
    # Set the data in the level-related objects.
    level.set(str(levelIndex) + "_level")
    lava.set(str(levelIndex) + "_lava")
    stars.set(str(levelIndex) + "_stars")
    # Reset the player's position...
    player.reset()
    # ...and all the navigation variables.
    leftPressed = 0
    rightPressed = 0
    xSpeed = 0
    ySpeed = 0
    # Change the state to the given level.
    state = levelIndex
```

Code 61: Setting the level.

Note that since the navigation variables are now set when the level is chosen, I can remove the lines setting them before the main loop. Also, this solution uses *set()* methods to make the objects import new data. This methods do not exist yet and therefore need to be added. This is mostly a matter of moving some variable instantiations from the constructors. *self.*z is also reset to zero, so that the levels always start from the same z position. For the *ThreeDMesh* it will be:

```
class ThreeDMesh():
    def __init__(self, baseColour, maxColour):
        self.z = 0
        self.baseColour = baseColour
        self.maxColour = maxColour
```

```
            self.currentColour = baseColour

    # Set the object to a given level.
    def set(self, id):
        # Reset the self.z attribute to start each level at the same z position.
        self.z = 0
        # Set the id.
        self.id = id
        # Import the data from a text file.
        self.data = self.importData()
```

And for *Stars*:

```
# The class that handles all things related to stars
class Stars():
    def __init__(self, baseColour, maxColour):
        # Set the attributes
        self.z = 0
        self.baseColour = baseColour
        self.maxColour = maxColour
        self.currentColour = self.baseColour

    # Set the object to a given level.
    def set(self, id):
        # Reset the self.z attribute to start each level at the same z position.
        self.z = 0
        # Set the id.
        self.id = id
        # Reset the score.
        self.score = 0
        # Import the data from a text file
        self.data = self.importData()
```

This allows the player to choose a level by clicking the corresponding tile.

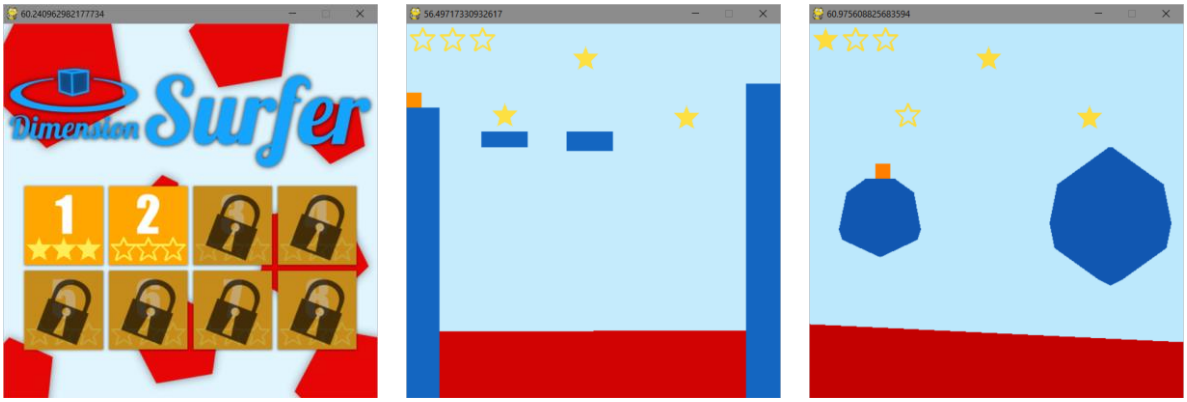


Figure 77: Clicking the two tiles results in opening two different levels (I have added the previously used collision testing environment as a placeholder level).

I have previously thought about adding a loading screen, but the loading time is unnoticeable.

Exiting to the main screen (2.4)

As indicated in the interview, there should be a way for the user to get to the main screen while playing the game. This might for example be to choose another level, and will be accomplished by pressing the escape key on the keyboard.

This can be done by adding and additional conditional to the level state processing loop:

```
# Event processing - we iterate on the events given to us by pygame:
for event in pygame.event.get():
...
    # Handle the keydown events.
    elif event.type == pygame.KEYDOWN:
    ...
        elif event.key == pygame.K_ESCAPE:
            # If the user wants to go to the main screen, set firstDraw to one
            # so that the main screen is drawn, and then switch state.
            firstDraw = 1
            state = 0
```

Code 62: Escaping to the main screen.

As an indication that this is possible, I added a subtle instruction to the main screen background:

Figure 78: The main screen background with added instructions for coming back to the level selection.

Beating the level (1.5, 4.8)

As defined in Winning conditions, beating the level is defined as reaching the right-hand side of the screen. This can be detected in the level state section of the loop by comparing the player's x coordinate with 500 pixels – the width of the screen. If it is in fact greater than 500 pixels, the game will change its state to -1 and show a "You Win" screen. For the detection part:

```
# Change state if player won.
if player.x >= 500:
    firstDraw = 1
    state = -1
```

Figure 79: Check whether the player crossed the right-hand side of the screen.

This will set the state to -1, so in the next iteration of the main loop the "You Win" screen will be rendered and the high score processed.

Implementing the "You Win" screen (1.9, 3.3, 4.8)

This screen will show the "You Win" message, along with the current score and the new score. This can be accomplished with the previously devised star-drawing procedures and some correctly prepared bitmaps.

I will use the *firstDraw* variable again since it would be a waste of computer power to draw the same screen multiple times, and I also would like to use some transparency in this screen, and continuously drawing a transparent image over itself would quickly turn it opaque.

To construct this screen I will use three images: a generic background and two textual images indicating the previous high score or that a new high score has been established (I decided against using pygame's text drawing procedures to ensure compatibility across many devices, which may not have the same set of fonts installed).

Figure 80: The images used for the "You Win" screen. The one on the left is the background, the two on the right are messages about the high score (the checkerboard indicates transparency).

Then I import the images as pygame objects:

```
youWinImage = pygame.image.load("images/you_win.png").convert_alpha()
newHighScoreImage = pygame.image.load("images/new_high_score.png").convert_alpha()
prevHighScoreImage = pygame.image.load("images/prev_high_score.png").convert_alpha()
```

Code 63: Importing the images.

The code that goes in the state == -1 part of the main loop is:

```
if state == -1:
    # Show the winning screen.
    if firstDraw:
        # Render the backgorund.
        screen.blit(youWinImage, [0, 0])
        # Check if the current high score has been beaten.
        if stars.score > scores[levelIndex-1]:
```

```
            # If yes, then draw the "New High Score" message.
            screen.blit(newHighScoreImage, [281, 267])
            # Change the stored high score to the current score
            scores[levelIndex-1] = stars.score
        else:
            # If the high score has not been beaten, render the
            # "Current High Score" message.
            screen.blit(prevHighScoreImage, [331, 267])
            # Render the current high score using stars
            # and the algorithm used for that on the main screen.
            for i in range(3):
                if i > scores[levelIndex-1] - 1:
                    stars.drawStar(screen, 350 + i * 33, 330, 1)
                else:
                    stars.drawStar(screen, 350 + i * 33, 330, 0)
        # If the next level is not unlocked (and in range), unlock it.
        if levelIndex < 8 and scores[levelIndex] < 0:
            scores[levelIndex] = 0
        # Save the scores to the scores.txt file.
        s = open("scores.txt", 'w')
        s.write(" ".join([str(x) for x in scores]))
        s.close()
        # Render the current score using stars
        # and the algorithm used for that on the main screen.
        for i in range(3):
            if i > stars.score - 1:
                stars.drawStar(screen, 54 + i * 33, 330, 1)
            else:
                stars.drawStar(screen, 54 + i * 33, 330, 0)
        # Refresh the screen
        pygame.display.flip()
        # Indicate that the screen has been drawn already.
        firstDraw = 0
    # The event loop must in this case be after the drawing part. If it wasn't
    # organised this way, setting firstDraw to one in the event loop would
    # trigger drawing the "You Win" screen instead of the main screen.
    for event in pygame.event.get():
        # If the event type is QUIT, the user wants to close the window.
        # So we set done to True.
        if event.type == pygame.QUIT:
            done = True
        # If any key is pressed or the mouse is clicked, we go to the main screen.
        elif event.type == pygame.KEYDOWN or event.type == pygame.MOUSEBUTTONDOWN:
            firstDraw = 1
            state = 0
```

Code 64: The ‘‘You Win’’ screen code.

The coordinates of the rendered elements were determined using Inkscape, which I used to design the screen.

I tried to complete the first level two times to see whether the system works. For the second time I also collected some stars in the second (testing) level to see if they will not be reset:

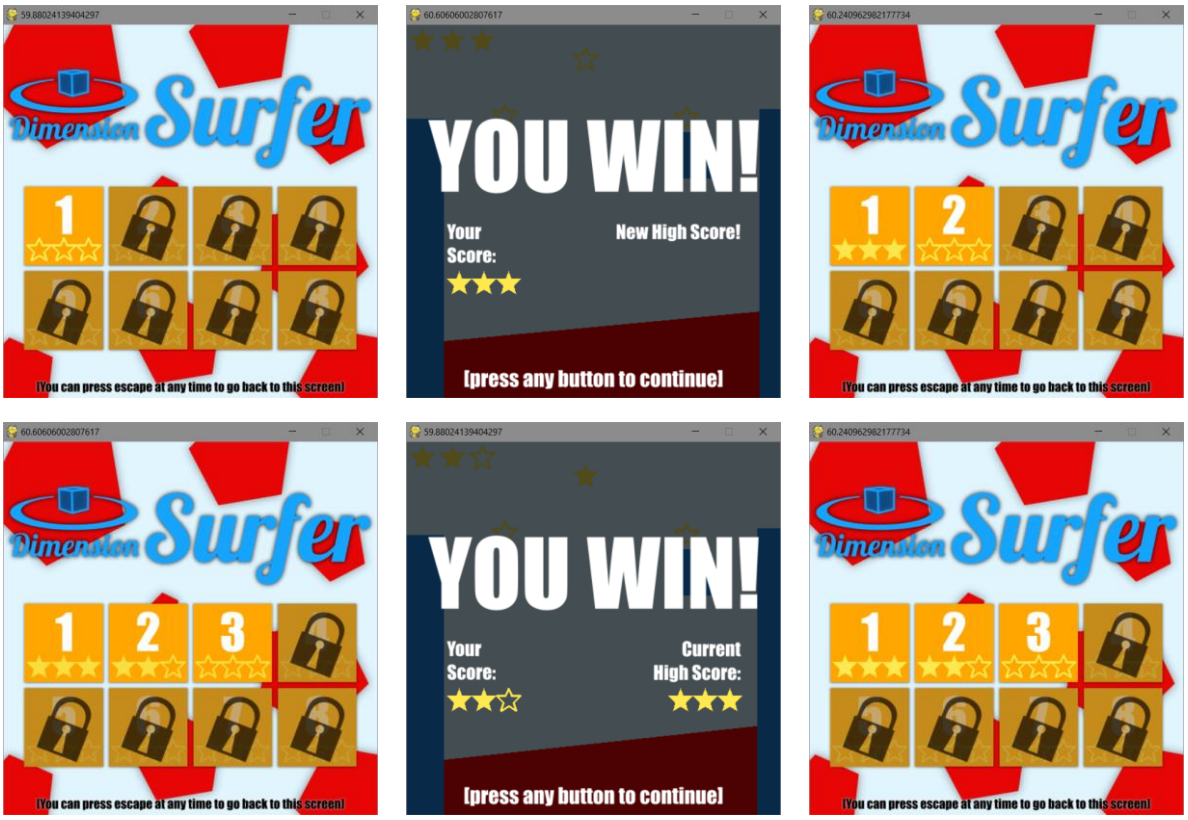


Figure 81: Testing the high score saving and level unlocking system in different cases. It successfully saves the correct high score, does not reset the score of the next level and shows the current high score if the new score is lower.

I also used the second (test) level to obtain different scores and see whether the score rendering works:

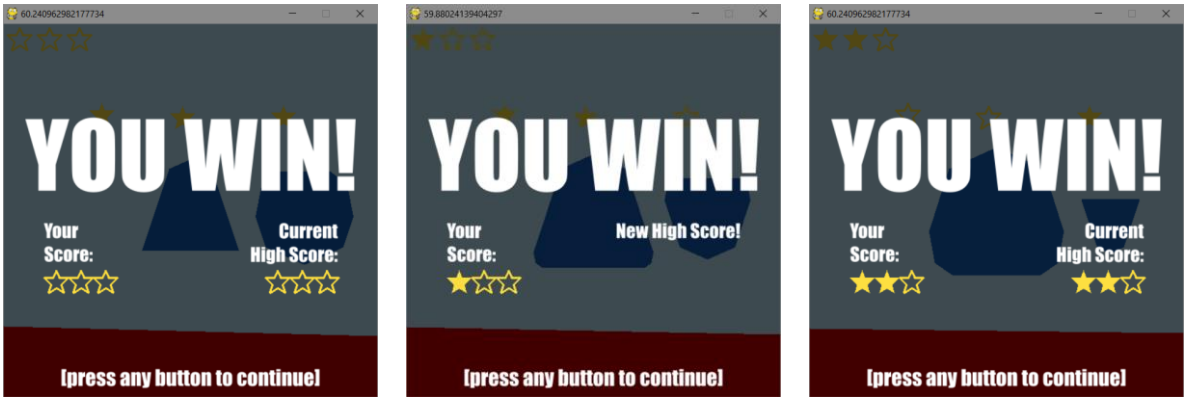


Figure 82: Testing the rendering of various star scores and high scores.

The tutorial (1.10)

Rendering the tutorial was discussed in Design section's Displaying the tutorial. Now I will create a class to handle this.

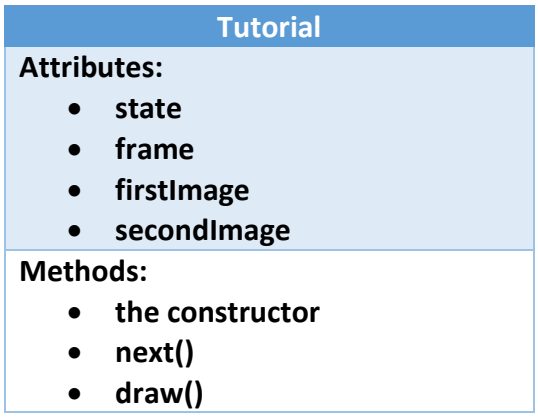


Figure 83: The class diagram for the Tutorial class.

First, I will create an instance of this Class before the main loop:

```
1.  tutorial = Tutorial()
```

Then I will add an additional clause to the level state event processing loop:

```
1.  for event in pygame.event.get():
2.      # If the event type is QUIT, the user wants to close the window.
3.      # So we set done to True.
4.      if event.type == pygame.QUIT:
5.          done = True
6.      ...
7.      # Go to the next tutorial screen.
8.      elif event.type == pygame.MOUSEBUTTONDOWN:
9.          tutorial.next()
```

And finally I will add a draw call at the end of the level state section, after drawing level, lava, stars and the player, but before checking whether the player had won:

```
1.  # Display the tutorial.
2.  tutorial.draw(screen)
```

The code of the class itself is:

```
1.  # A class for displaying the tutorial
2.  class Tutorial():
3.      def __init__(self):
4.          # Set the state and current frame to zero.
5.          self.state = 0
6.          self.frame = 0
7.          # Load the images
8.          self.firstImage = pygame.image.load("images/wsad.png").convert()
9.          self.secondImage = pygame.image.load("images/animationsheet.jpg").convert()
10.
11.     # Change to the next state
12.     def next(self):
13.         self.state += 1
14.
15.     # Draw the image corresponding to the state
16.     def draw(self, screen):
17.         if self.state == 0:
18.             # Draw the image explaining the use of the WSAD keys
19.             screen.blit(self.firstImage, [0,150])
20.         if self.state == 1:
21.             # Draw the animation explaining the concept of the third dimension.
```

```
22.             # Increase the frame counter.
23.             self.frame += 1
24.             # Calculate the frame to render.
25.             renderFrame = (self.frame//8)%24
26.             # Cut the animation sheet according to the renderFrame variable
27.             # and render it on screen. The third argument are the coordinates
28.             # and dimensions of the cut
29.             screen.blit(self.secondImage, [0,150], [0, renderFrame*200, 500, 200])
```

Code 65: The Tutorial class.

The images used in this class (*wsad.png* and *animationsheet.jpg*) were specified in the Tutorial section. I created the animation in Blender (the 3D graphics software that I also used for creating the levels), and then I used the free version of TexturePacker by CodeAndWeb GmbH[8] to join the frames into an animation sheet which I could use in the game by cropping it to extract the frames. I used the .jpg file format to save this image (instead of .png used for all the other graphics) since I did not need transparency and since the dimensions of the animation sheet are very big, the file size was too great in comparison to the size of the other files.

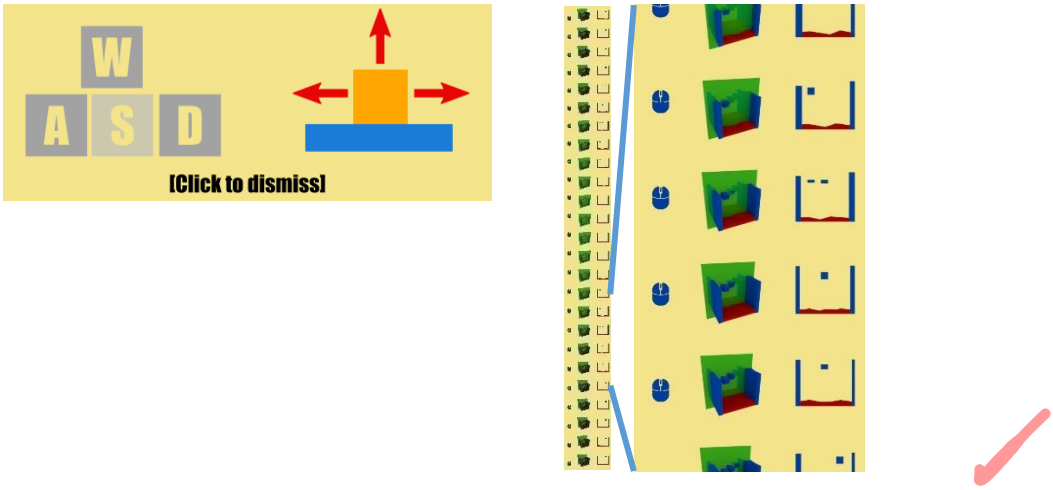
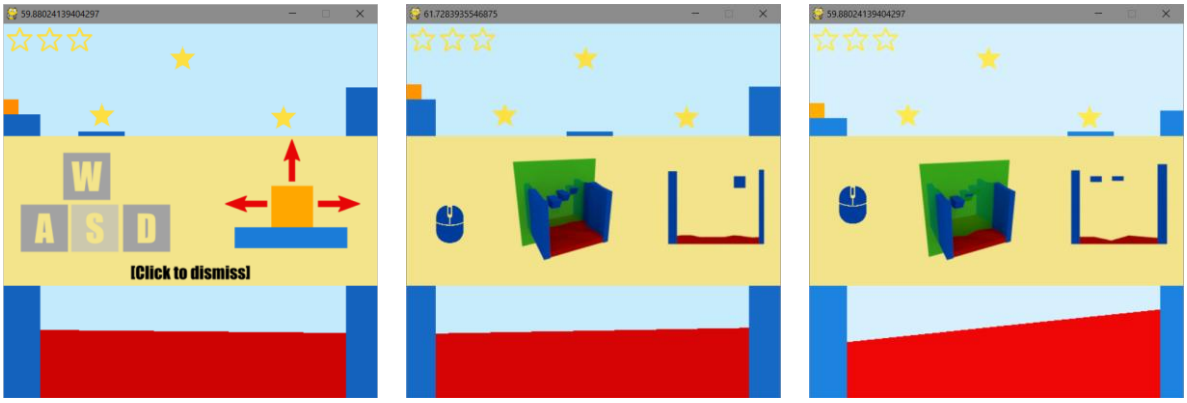


Figure 84: The first tutorial screen, the animation sheet used for the second screen and an enlarged fragment of it.

This results in correctly showing the tutorial at the beginning of the level which is chosen by the player straight after starting the game. It does not show later and is dismisible by a mouse click, as intended. Here are some screenshots of the tutorial in operation:



[8] https://www.codeandweb.com/texturepacker

lots of detail about implementing the various usability aspects of the project

Figure 85: The tutorial working. The animation is played correctly, I only screenshotted two of the frames.

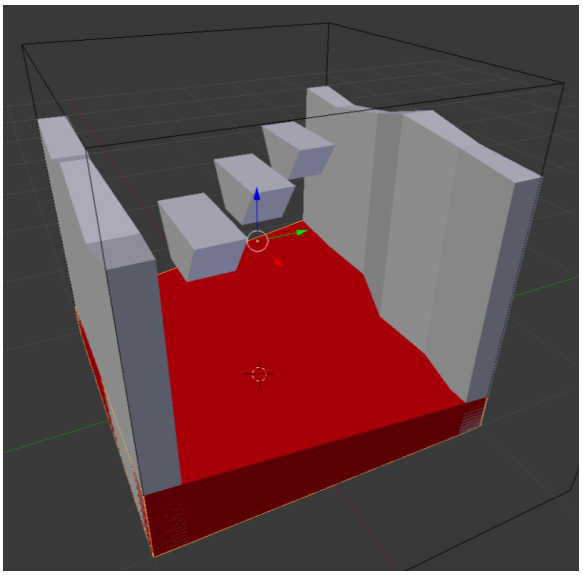
Creating the levels (1.1)

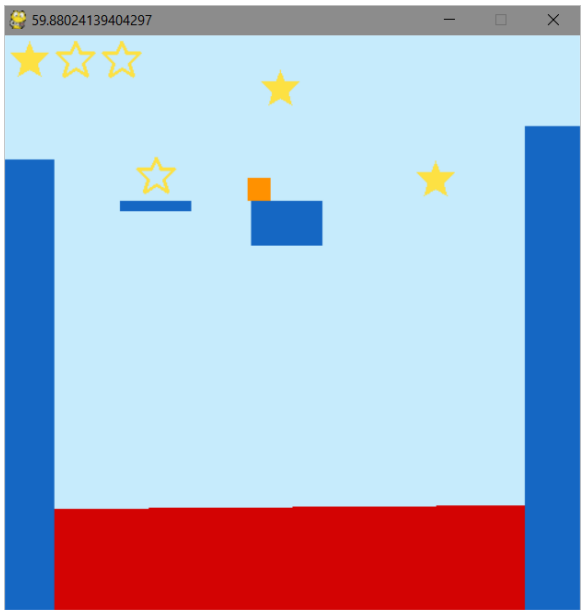
Now that the game is practically ready, I need to create the 7 remaining levels for it. I will use Blender for that and uncomment the

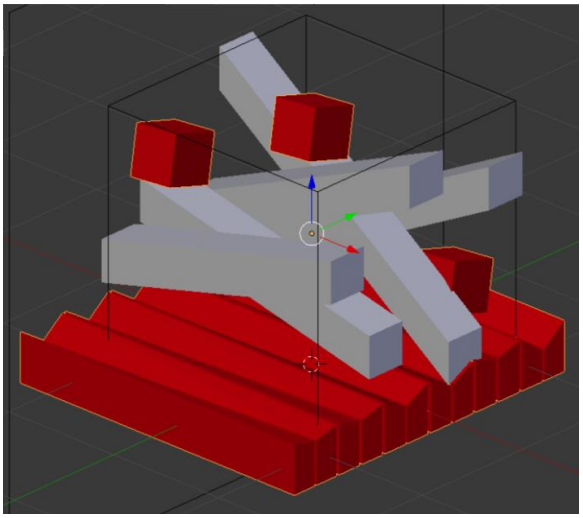
```
1.  print(mouse_x, mouse_y)
```

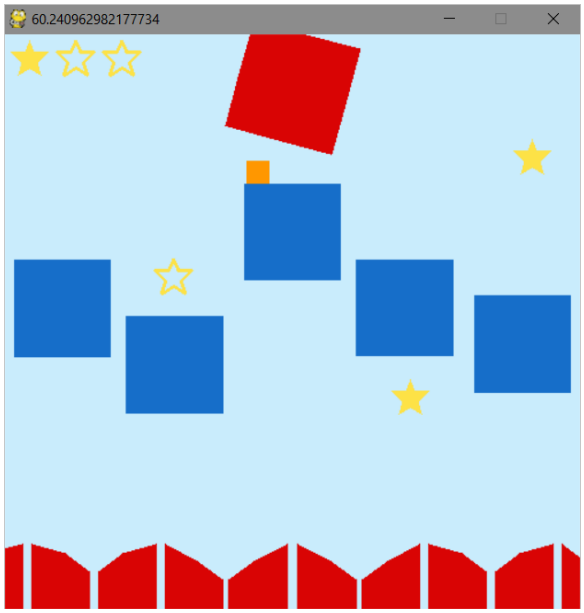
line mentioned in Storing the star data.

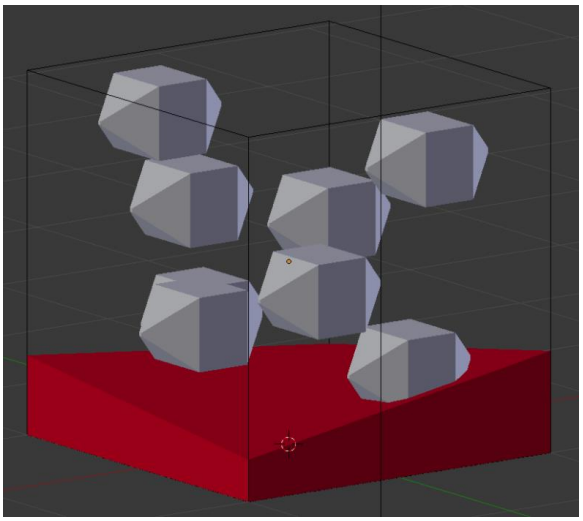
I created new levels for the game in Blender and added stars based on the printed mouse x and y coordinates. I also decided to use the level created for gaining key user feedback as the second level. Below are screenshots of levels 1, 2, 3, 4, 5, 6, 7 and 8 from Blender (3D) and one frame from the game for each level (2D).

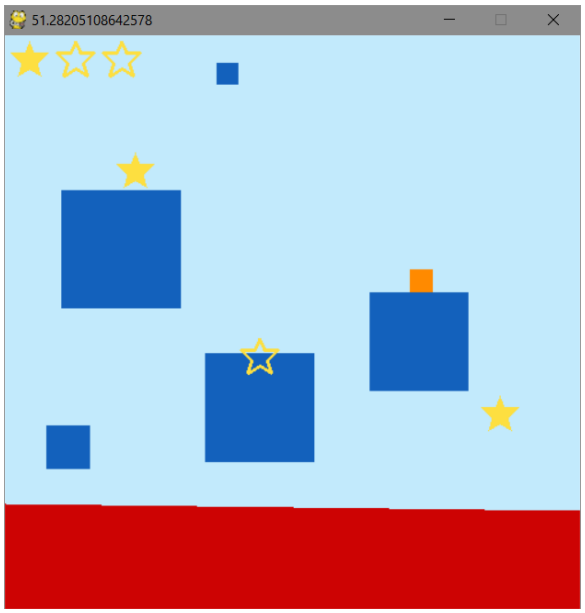


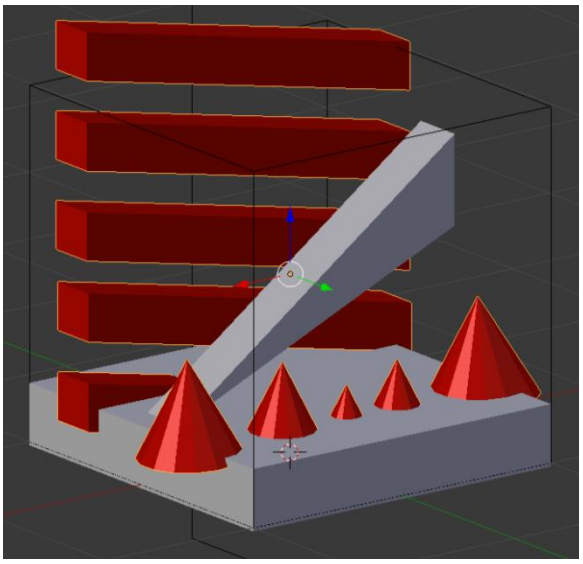


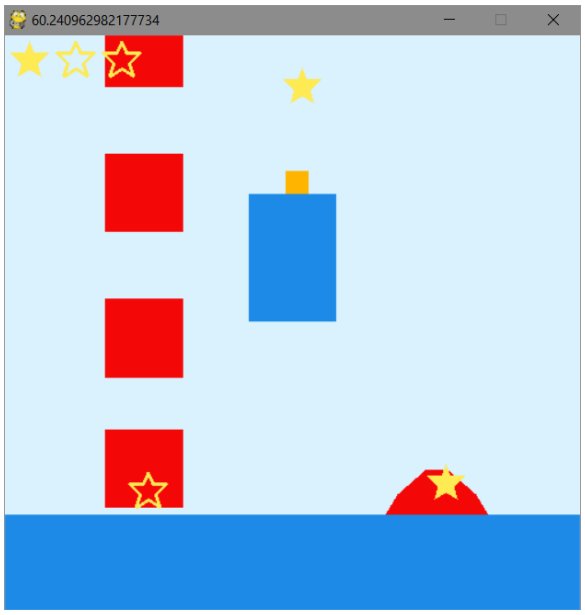


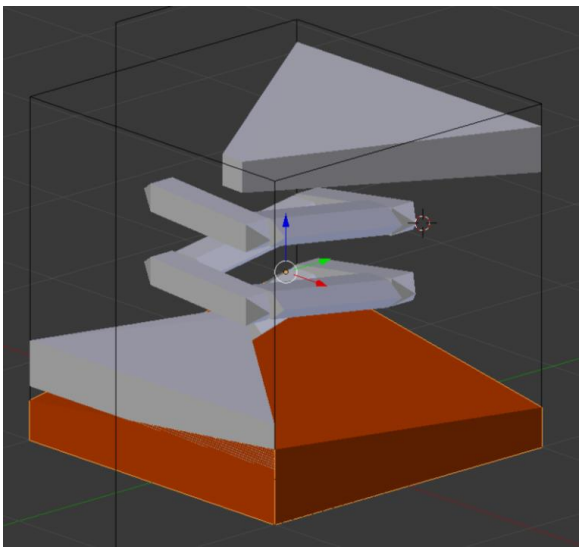


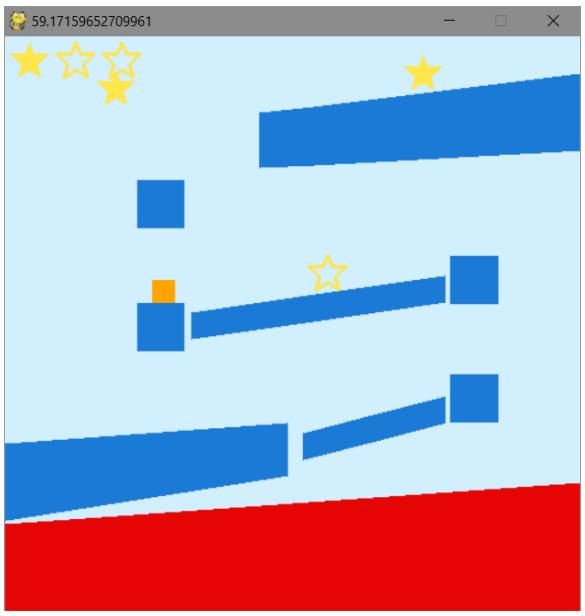



51.28205108642578




60.240962982177734




59.17159652709961

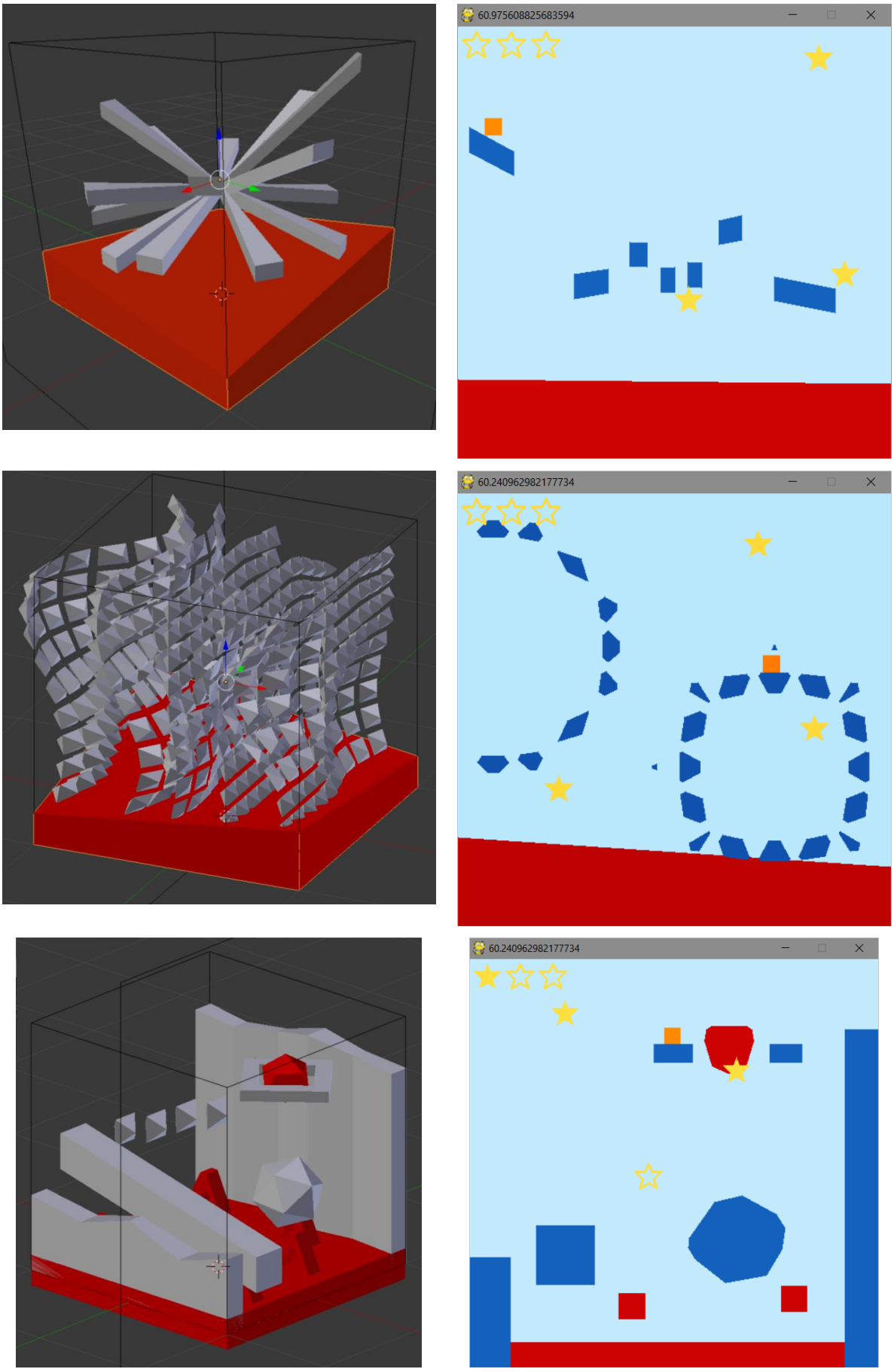


Figure 86: Levels 1, 2, 3, 4, 5, 6, 7 and 8 in 3D and 2D format.

All levels are playable and their level of difficulty increases progressively.

This concludes the implementation stage.

Development 15/15 There is plenty of evidence here of an iterative development process that has broken down the problem as it is related in the analysis stage. The algorithms continue to be numbered from the earlier stage for ease of reference.
The iterative development has been informed by ongoing testing and prototyping as is seen from the various screenshots.
The code is modular in nature and well laid out with annotation throughout. Data structures and variables are shown along with ongoing review and refinement at all stages

Testing to inform development 10/10 A lot of evidence to show ongoing testing at each stage. Where tests have revealed required corrections and refinements these are shown along with full justification explaining the approach taken.

Evaluation

Now that implementation is finished I can evaluate how well the project has been executed and analyze its further possible development and maintenance.

Testing

The testing has been largely performed and described in the Implementation stage. I also tested the game against the success criteria devised in Requirement specification, and the results can be found in Testing against requirements and success criteria. Generally, my solution fulfills the requirements very well, and I managed to satisfy every point of the success criteria. Additionally, I have played the game a lot to look for possible errors appearing during the gameplay, but I found none.

As an additional test, I tried to test the game's robustness by pushing it to its limits in different ways:

Test	Expected behaviour	Actual behaviour
Clicking the mouse in random places on the main screen.	If the mouse cursor is not over the level tiles, the condition implemented on page 103 should ensure that a random level is not chosen.	Nothing happens when the player clicks outside of the level tiles.
Trying to choose locked levels.	The code on page 104 should check whether the level is locked and do nothing if it is.	Clicking a locked level tile does nothing.
Pressing random keys during gameplay.	Keys that are not WSAD, space or arrow keys are not handled in event loops, they should do nothing.	Pressing random keys does not yield any reaction.
Moving mouse around very quickly during gameplay.	The damping mechanism implemented on page 68 should prevent the level from changing rapidly.	Rapid mouse movement results in the level changing smoothly.
Beating all the levels.	A condition implemented on page 108 should stop the game from trying to unlock level 9, which does not exist.	When all levels are beaten, the game does not crush, meaning that the condition worked.

Switching focus to another window	The game should continue running and when focus returns, the easing mechanism should prevent the level from changing rapidly due to sudden changes in mouse position.	Switching focus to other windows and back, as well as moving the mouse cursor outside of the game's window and then returning in another place does not cause any unexpected behaviour.
Trying to play the game on a very slow computer.	Python is cross-platform, so it should work on any desktop computer. Performance might be a problem.	The game runs very slowly on my Raspberry Pi Zero, details below.

All the tests succeeded, except for the last one. I tried to run the game on my Raspberry Pi Zero (1GHz, Single-core CPU, 512MB RAM), which is a budget computer definitely not designed to run games.

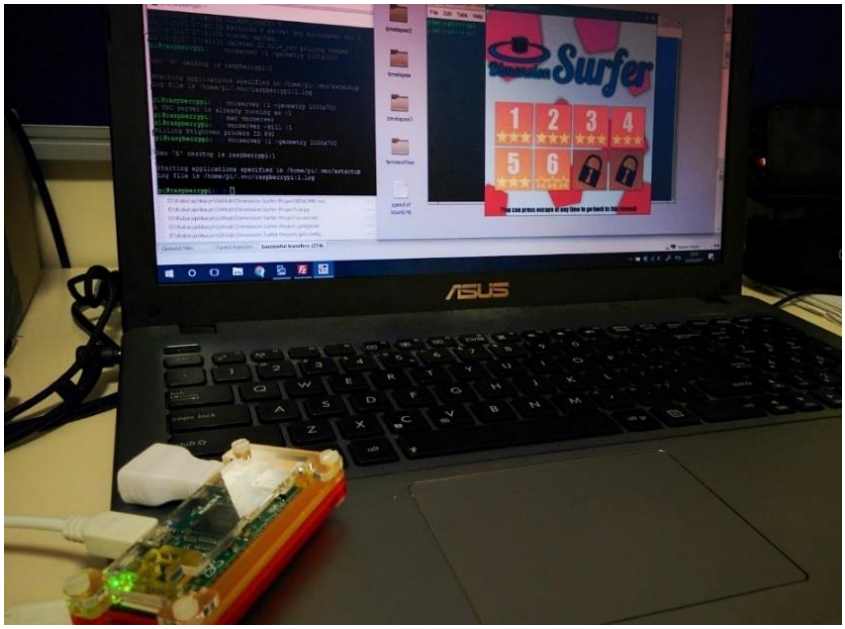


Figure 87: The game running on my Raspberry Pi Zero, using VNC (Virtual Network Computing) to display the screen on my main laptop.

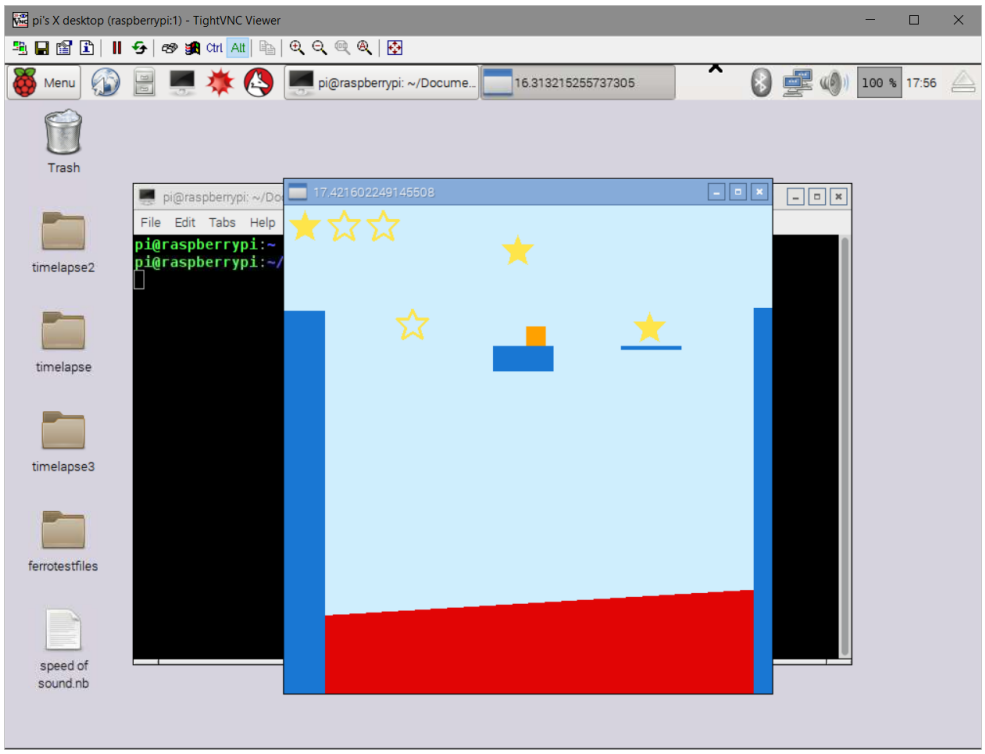


Figure 88: The game running on the Raspberry Pi Zero.

Although the game successfully launched and performed well (around 60 frames per second) in the main screen and the "You Win" screen, the performance decreases significantly when a level is launched – the framerate drops to around 10 to 15 frames per second.

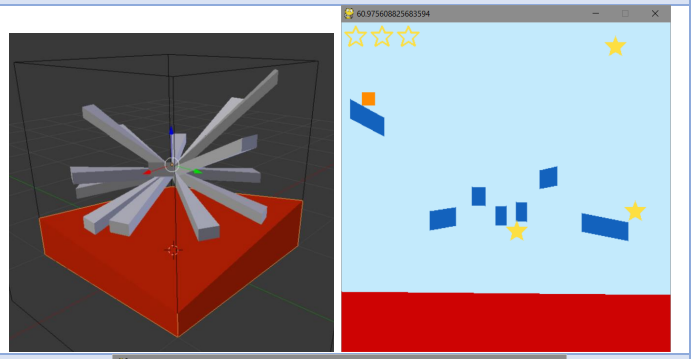
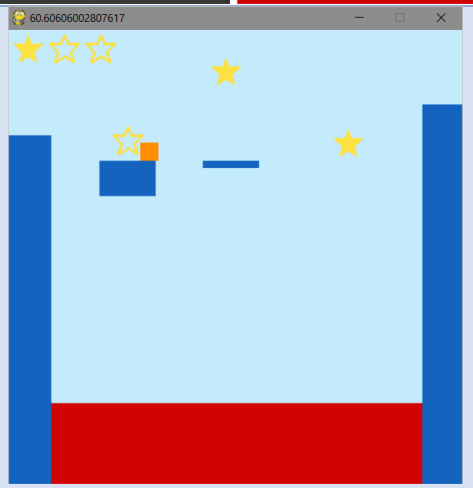
Initially I anticipated that due to rather small hardware requirements for Python ad pygame the game should run without problems on computers like the Zero. It turns out that this is not true. This seems to be mainly due to the lack of hardware graphics acceleration on the Pi. On the Pi the processor is used not only to calculate collisions, but also to draw the graphics, which increases the load greatly and decreases performance. That means that on more popular computers (that do have hardware acceleration) it should work just right.

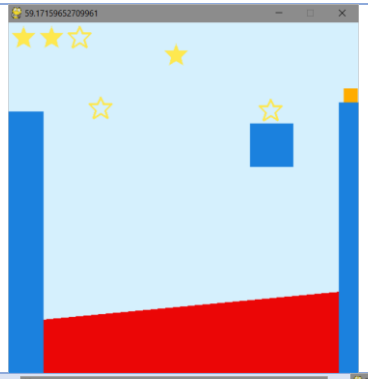
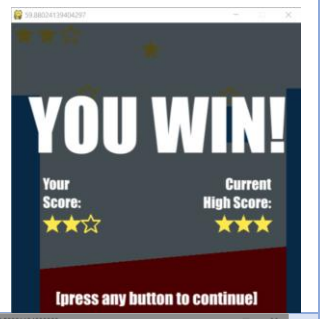
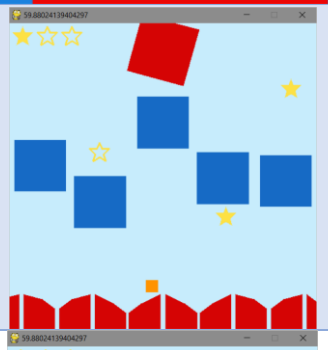
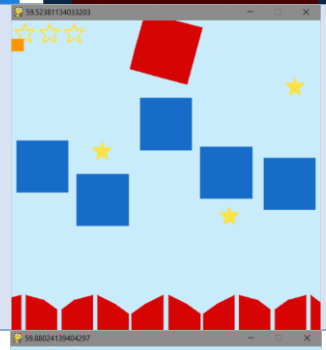
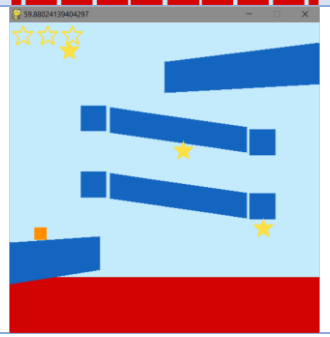
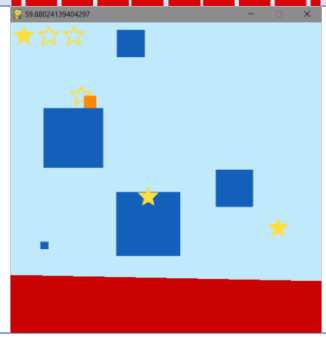
To test that conclusion I tried the game on other computers and there are no performance problems.

Testing against requirements and success criteria

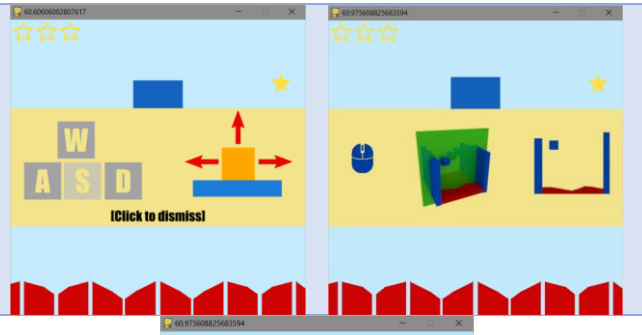
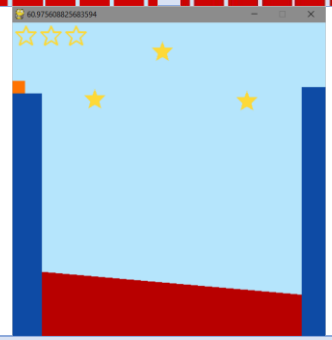
To test thoroughly whether I have met the criteria I took each and tested whether it works as expected. The results (with screenshots as proof) are presented in the table below.

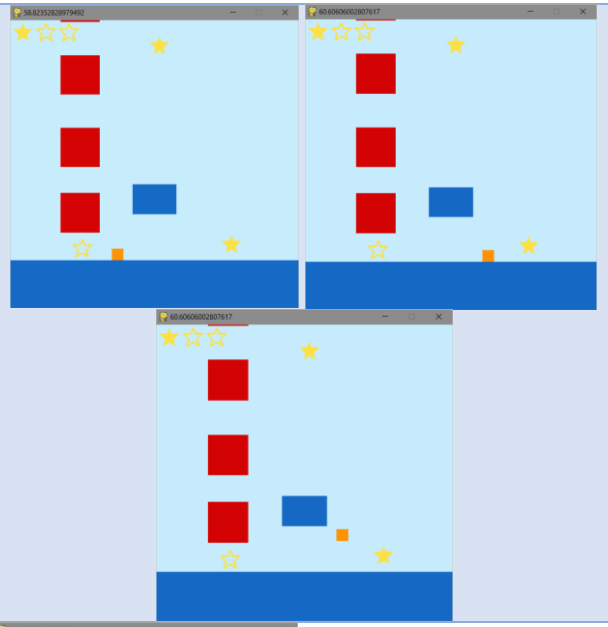
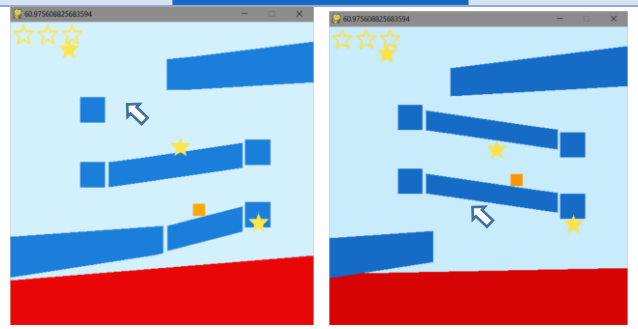
The success criteria are again highlighted with a bold font.

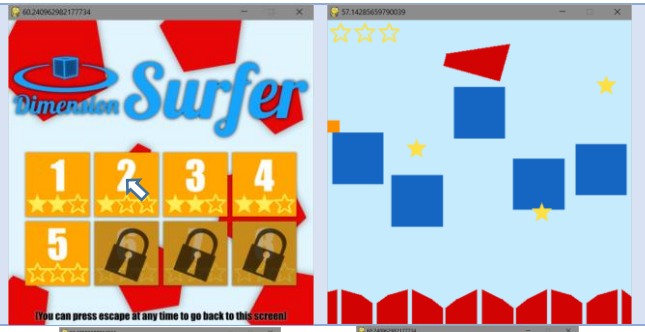
Success criteria	Fulfilled?	Comments	Screenshots
1. Design requirements			
1.1. The game's levels are created as 3D meshes and presented to the player as 2D cross-section.	Yes	I create the levels in the 3D software Blender and what the player sees are cross-sections of it.	
1.2. Stars placed in various areas of the level to be collected by the player.	Yes	Stars are placed in different places across the levels and are collectible.	
1.3. Player's sprite is a square.	Yes		
1.4. A modern-looking colour scheme that also plays on the classic colour associations: orange player, blue obstacles, yellow stars, red lava surfaces.	Yes	The colour scheme was applied. In addition, it was determined that it is suitable for people with colour blindness.	

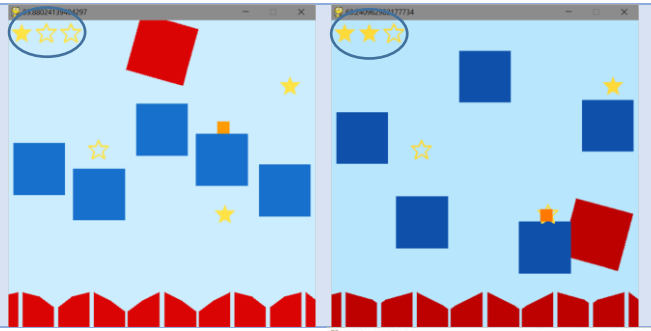
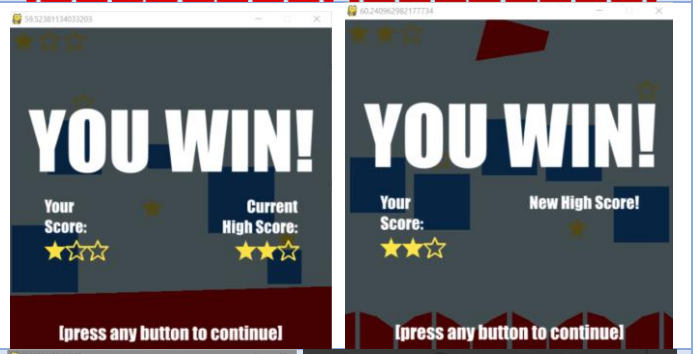
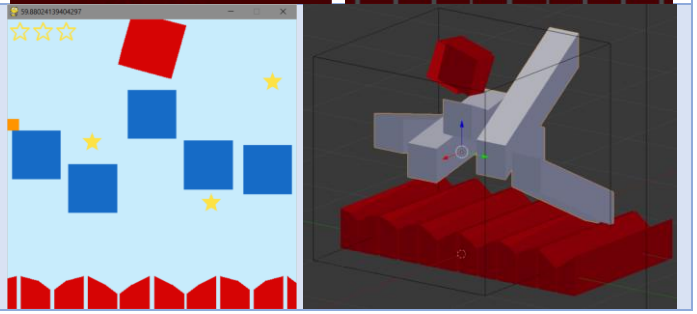
1.5. **Winning the level requires the player to get to the right-hand side of the screen.**	Yes	Moving to the right displays the "You Win" screen.	 
1.6. **Touching a lava surface results in player's death – return to the level beginning and losing the stars.**	Yes	Colliding with any lava surface successfully restarts the level (the star score is set to zero and the position of the player to the left top corner of the screen).	 
1.7. Lava surfaces used to prevent the player from falling out of the level.	Yes	Lava used as the bottom layer of many levels.	 

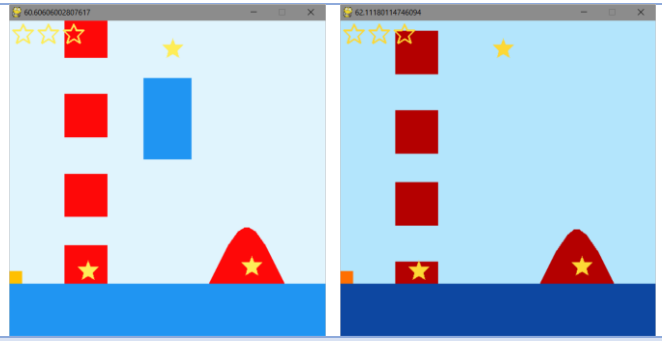
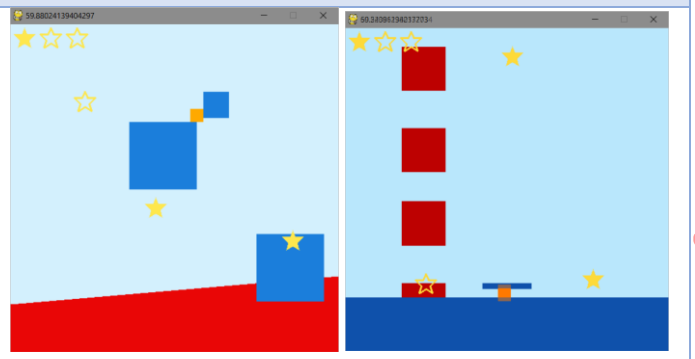
1.8. Levels can be chosen from a starting screen.	Yes	The start screen shows unlocked levels and enables player to choose one of them.	
1.9. Player unlocks access to a level by beating the level before it.	Yes	The next level is unlocked (it can be chosen from the main screen) when the previous one has been finished.	

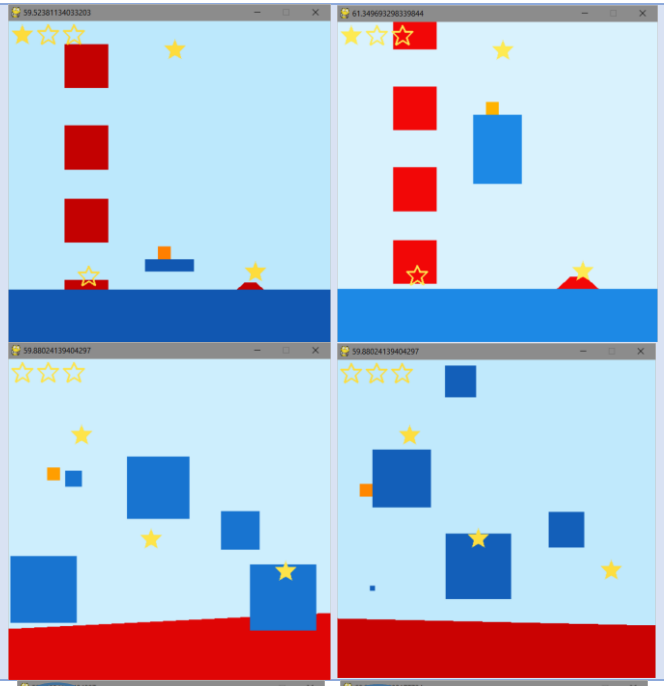
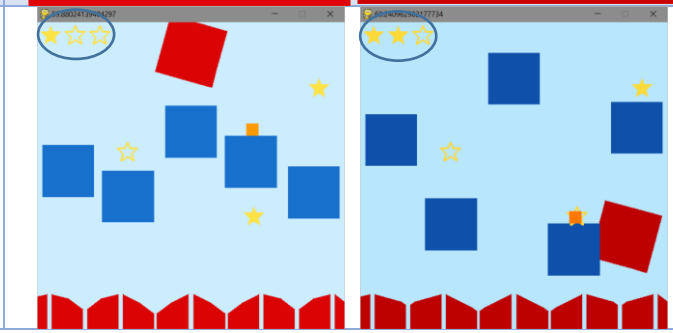
1.10. There is a tutorial at the start of the first level chosen by the player to introduce the player to the basics of input.	Yes	The tutorial is shown after the user starts the game and chooses any of the levels, it is animated properly and instructive.	
1.11. The first level is designed so that it naturally helps the player understand the game's concepts.	Yes	The first level (in addition to the tutorial) shows that the purpose of the game is to get to the other side of the screen, stars are collectible, lava kills and the geometry of the level must be changed in order to progress.	
1.12. There are eight levels in total.	Yes	The eight levels are displayed in the Creating the levels section.	
2. Input requirements			

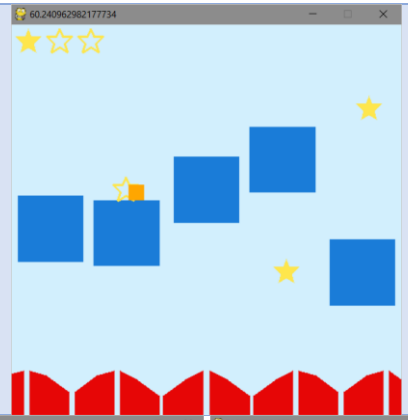
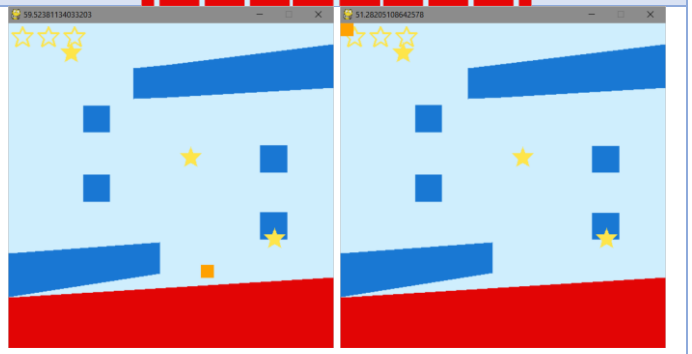
2.1. WSAD keys or arrow keys used to control the player.	Yes	The player can be moved by using ‘a’ and ‘d’ keys as left and right, ‘w’ and space make the player jump, using the arrow keys also works as an additional option.	
2.2. Mouse used to control movement in the third dimension.	Yes	By moving the mouse up and down the user can change the displayed cross section and thus change the geometry of the level.	 The arrow represents the position of the cursor.

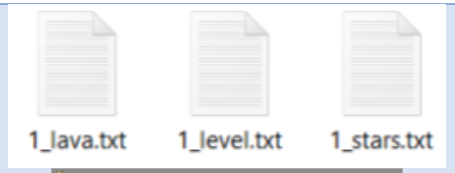
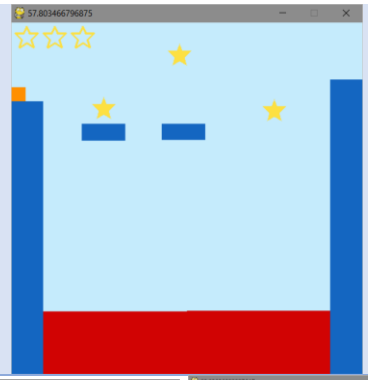
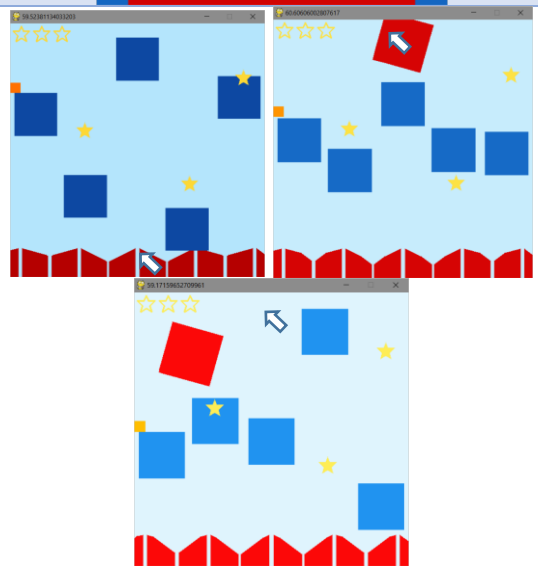
2.3. Mouse used to choose a level from the main screen.	Yes	By moving the mouse to one of the unlocked tiles and clicking it the user is able to choose a level to be launched.	
2.4. By pressing the esc key player returns to the main screen.	Yes	By pressing the escape key the player can go back from any screen to the main screen.	
3. Output requirements			
3.1. Main game screen shows which levels are unlocked and high scores for each level.	Yes	The high scores are shown on each of the tiles and the tiles corresponding to the locked levels are covered with locked icons.	

3.2. **A star score is shown throughout the game play and stars appear empty when collected**	Yes	The star score is displayed in the top left corner of the screen and updated as stars are collected, along with the appearance of the collected stars.	
3.3. **After finishing the level, a screen is shown displaying the high score and the current score.**	Yes	The "You Win" screen displays the obtained score and the high score. If the score obtained is higher than the high score a "New high score" message is displayed.	
3.4. **A cross-section of the three-dimensional level is rendered.**	Yes	Cross sections of the platforms (blue) and lava (red) are displayed on screen. This can be verified by comparing with a cross section generated directly in Blender (3D graphics software).	

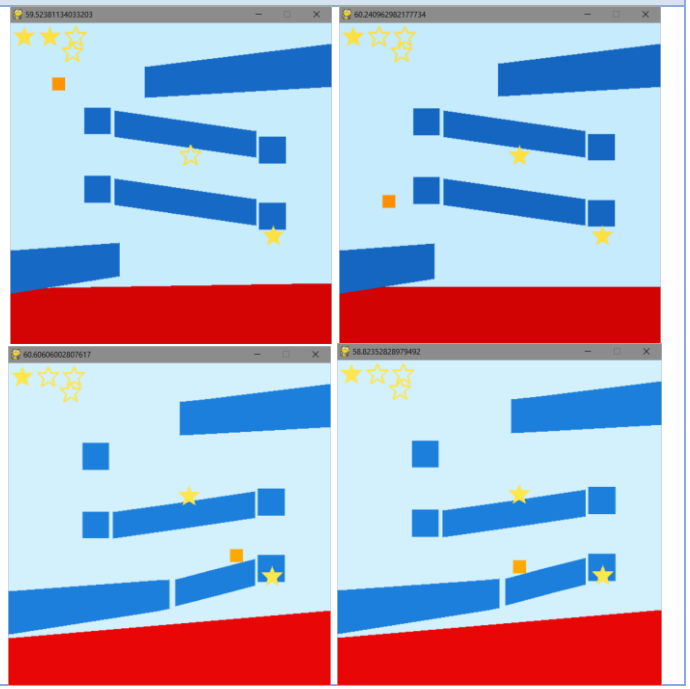
3.5. Colours of the game elements change from lighter to darker with the changes in the third-dimension to provide simple information on position.	Yes	As different cross sections are chosen, the colours of the entire level change from light to dark.	
4. Processing requirements			
4.1. Player should not be able to go through obstacles.	Yes	The collision system works without noticeable artefacts (tested by playing the game a lot and getting into weird collision situations). It can handle collisions with multiple objects and conflicting projection vectors (when squashed for example).	 On the second screenshot the player oscillates between two positions instead of being thrown away randomly, which sometimes happens in other collision systems.

4.2. Changes in level shape should not result in random behaviour of the player's sprite.	Yes	As above (4.1), squashing does not make the player move randomly. Also, when the level changes the player is displaced correctly.	
4.3. Colliding with a star should add one to the star score.	Yes	When the player overlaps with the star, one is added to the score.	

4.4. Stars that were once collected should not be collectible again.	Yes	When the player goes again to a star that was already collected, the star score does not increase and the does not change state.	
4.5. Colliding with a lava surface results in resetting the level.	Yes	Collision with a lava surface results in setting the score to zero and placing the player in the top left corner of the screen.	 Since the collision moment itself is hard to capture, I took the screenshot right before it, when the collision is already inevitable.

4.6. The data about the level is imported from text files and properly processed.	Yes	The data files are chosen based on the level selected and their contents interpreted correctly, resulting in displaying the level.	 
4.7. Mechanisms are introduced to make the movement in the third dimension smoother.	Yes	The changes of the cross sections do not follow the mouse movement directly, but are smoothed out algorithmically.	 The cursor is moved rapidly to the top of the screen, but the level follows there slowly, reaching the desired state after around two seconds.

4.8. Crossing the right edge of the screen should result in winning the level, saving a high score and unlocking the next level.	Yes	Crossing the right edge of the screen results in showing the "You Win" screen (shown in 1.4). This also results in changing the *scores.txt* file to update the high score if needed and unlock the next level if it has not been unlocked before.	Let us start from these contents of *scores.txt*.
			`3 1 2 0 -1 -1 -1 -1`
			Now I have played level four. New high score is saved and level five is unlocked:
			`3 1 2 2 0 -1 -1 -1`
			Now after playing level 2 again I get a new high score, but the score in level 3 is not altered (not set to zero):
			`3 2 2 2 0 -1 -1 -1`

4.9. The player's sprite should fall under simulated gravity.	Yes	When the player is unsupported by a platform, he starts falling. Additionally, when the platform is inclined, the player starts sliding down, with the pace depending on the steepness of the surface.	

As shown in the table I managed to meet all the success criteria, as well as all the other, non-critical requirements.

Testing to inform evaluation 5/5 There is full annotated evidence of post development both for functions and robustness and usability

Additional detail of the iterative approach, testing and remedial steps taken

Changes made in the development process

Due to the iterative nature of development and testing the game at every stage of implementation, some of the features were not programmed exactly as designed, although they achieve the same results as in the approaches proposed in Design. Here is a short list of the changes I made to the design in the Implementation stage:

- Introduced an object-orientated coding approach.
- Added an option to print mouse coordinates to add stars more easily.
- Created a separate library for the Separating Axis Theory Functions
- Outlined the differences between *Lava* and *Level*, putting the common features into a parent class (*ThreeDMesh*)
- When a sign change needed, used Python's *copysign()* function instead of dividing by absolute value to avoid division by zero.
- Introduced a condition that omits elements that are too small when checking for collisions.
- Introduced a condition that makes sure the x and y axes are not checked twice.
- Introduced a y speed limit.
- Introduced a trigonometric method of reducing speed on slopes based on the incline of the slope.
- Changed the star rendering mechanism from rendering bitmaps to polygons to enable changes in colour.
- Disabled the possibility to fall out of the left side of the screen.
- Introduced an improvement to the collision system, making collisions with more than one object at a time.
- Added instructions for escaping to the main screen on the main screen.
- Introduced a mechanism of rendering the static screens only once.

Those changes solved errors or increased the performance.

Usability features

Additional detail about usability features development

These are the main usability features of the game:

- The level can be chosen from the main screen.
- There is a tutorial when the user first chooses a level explaining navigation.
- There are instructions on how to return to the main screen and dismiss the tutorial.
- The first level easily introduces the player to the main concepts of the game.
- High scores and scores are shown on the main screen and during gameplay.
- The colour scheme is suitable with colour blindness.

Most of these were already tested while checking against the success criteria. To test whether the game is suitable for people with colour blindness, I put two screenshots from both ends of the colour ranges specified in Colour scheme and applied a special filter[9] to them:

[9] http://www.color-blindness.com/coblis-color-blindness-simulator/

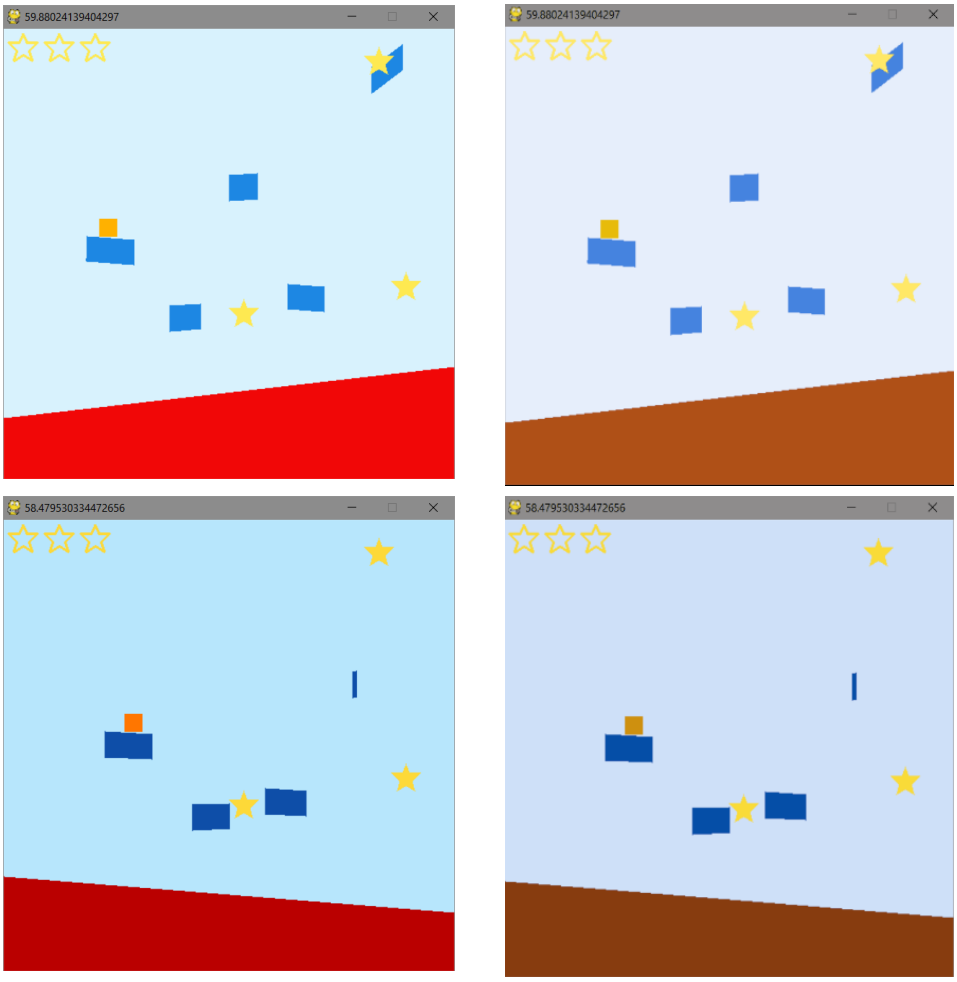


Figure 89: Simulation of colour blindness. The results shown are for protanomaly, the most common type of colour blindness, but I tested all of the other types and people with them also would be able to distinguish between objects without problems.

I also gathered some feedback about this features from my key user, which is outlined in Key user end interview.

Possible Future Improvements

Although I have met all the specified requirements, the game obviously is not perfect. As with any project, improvements can be made. This could include creating more levels, introducing character graphic for the player sprite, a more dynamic background and more complex graphics in general and add a storyline so the game is more engaging. I could also create a version with levels created by consulting with Mathematics teachers. This version could be used in lessons to explain higher dimensions with examples and train students' ability to imagine complex 3D shapes.

Another area of development could be porting the game to mobile platforms. The current code should work without problems on any major desktop platform, but Python is not directly available for Android and iOS. I did find a project that claims to successfully port pygame games to Android installable packages (RAPT: Ren'Py Android Packaging Tool[10]) and I believe

[10] https://github.com/renpytom/rapt-pygame-example

that this is worth investigating. The mobile gaming market is much more open to small, inventive games and could prove to be a natural environment for my game.

Maintenance

The first type of maintenance is of course correcting bugs that may be noticed by the end users. A bigger user base might be able to spot mistakes that I or my key user have not managed to spot. Those errors could be quite easily corrected thanks to the modular nature of the game (i.e. the *main.py* and *sat.py* files can be updated independently without influencing the stored high scores and unlocked levels). Users could report those issues through GitHub's issue tracker for example, which is a versatile feedback platform that allows discussion with the user and helps pinpoint the case of the problem and easily track the changes needed to repair it with the git version control system.

Since some parts of the game (especially the collision system) are developed by me (unlike external libraries that are written and checked by many people), the solutions that I used might not be optimal in all cases. If I find out that there is a better way to do something algorithmically, I should incorporate the improved solution to the game.

Another kind of maintenance is developing new levels, so that the game is constantly engaging for end users (if they finish all the available levels, there would be a possibility of more levels arriving). Distributing the new levels would require a special mechanism and changes in code, since now the main screen can only display 8 level tiles and the high score saving file and mechanic is constructed to handle only 8 high scores.

The maintenance is made significantly easier by the git version control system (described in detail in The programming setup). It allows me not only to track issues noticed by the users, but also to see the changes I make in the code organised and managed in a logical way.

Key user end interview

After assessing the fulfilment of the requirements myself, I showed the game to my key user to give me feedback. I asked him a few questions afterwards and these are his answers.

First of all, how do you like the game?

I think it's great! It improved a lot since you showed me the first prototype. The user interface elements make it easier to navigate and it is way more satisfying to beat the level when you get feedback on it and can beat your high scores. The tutorial was definitely a good idea and you executed it well – it is easy to understand and gives a better understanding of what the game is about. The new levels are definitely a highlight, they really show what is possible with the engine you created. I'm also glad you did introduce the improvements I suggested.

I have shown you the success criteria I listed based on our first interview, do you think I met them?

Yes, all of it seems to be fulfilled very well. Even if there is room for improvement, I think that at the current stage the game is ready to be presented to wider audiences.

Do you find the levels challenging enough?

Yes, they are interesting and hard, but not impossible. They also explore some weird and interesting ways in which the mechanism that you created can be utilised.

Are the game's usability features good? What I mean is whether it is easy to use, the on-screen help gives enough information to get to understand how to play it etc.

Yes, in my opinion the game explains itself quite well. You get the textual help for basic actions, but the game mechanic is described with pictures and animations, which is in my opinion the best way to do it, since it is easier to imagine based on them, not a piece of text.

Do you think that the game is suitable for use in education? I thought about using it as a way to present the students with examples concerning higher dimensions, stimulate their imagination and possibly even teach them how to extrapolate the concept to higher dimensions.

That's brilliant! I for one would really enjoy being educated through gaming, I think it makes you more engaged and interested. In this case the mathematical problem is rather abstract, so I think that presenting it in this format could be incredibly helpful.

Great. Thank you for working with me throughout this project, your feedback was really helpful for me.

It was a pleasure for me, it's great to take part in creating something like this. Thank you for inviting me to help you!

Evaluation of solution 15/15 - the test evidence is matched with success criteria to show criteria have been fulfilled. Further comments are given to indicate how future modifications might be implemented along with maintenance features.

The line of reasoning throughout the project is continued into this final stage which is clear and logical. All information given is relevant and substantiated.

Bibliography

- https://www.python.org – website of Python, the programming language I used.
- http://www.pygame.org – website of pygame, the Python library I used to develop the game.
- http://programarcadegames.com/ – an Internet course in pygame that I used to get started with the library.
- http://www.metanetsoftware.com/technique/tutorialA.html – a tutorial that I used as the main resource during developing the collision detection system based on the Separating Axis Thorem.
- https://github.com/z-gora/Dimension-Surfer-Project - the GitHub repository that I used during implementation.
- https://inkscape.org – Inkscape, the tool that I used to create graphics for the main screen and the 'You Win' screen.
- https://www.blender.org/ – Blender, the tool that I used to create the levels and slice them into cross-sections.

Appendix A: The file structure

Folders are indicated with a **bolder** font.

- **main folder**
  - main.py
  - sat.py
  - scores.txt
  - **images**
    - animationsheet.jpg
    - locked.png
    - main_background.png
    - new_high_score.png
    - prev_high_score.png
    - wsad.png
    - you_win.png
  - **level_data**
    - 1_lava.txt
    - 1_level.txt
    - 1_stars.txt
    - 2_lava.txt
    - …

Appendix B: The sat library

sat.py:

```
1.  # The Separating Axis Theorem library by Jakub Dranczewski.
2.  # Developed for the Dimension Surfer game, as part of a
3.  # Computer Science Project.
4.
5.  import math
6.
7.  # Project a given polygon onto an axis.
8.  def project(polygon, normal):
9.      # Create a list of projected vertices.
10.     projected = []
11.     # We treat each vertex coordinates as a position vector
12.     # and iterate on them.
13.     for vect in polygon:
14.         # Calculate the dot product of the position vector and the axis vector.
15.         dp = vect[0] * normal[0] + vect[1] * normal[1]
16.         # Calculate the projection of the position vector on the axis.
17.         projected_v = [normal[0] * dp, normal[1] * dp]
18.         # Calculate the projection's length - this is what we actually need.
19.         projected_l = math.sqrt(projected_v[0] ** 2 + projected_v[1] ** 2)
20.         # Get the direction of the projection relative to the axis direction.
21.         sign_p = projected_v[0] * normal[0] + projected_v[1] * normal[1]
22.         # Apply the direction to the projected length.
23.         projected_l = math.copysign(projected_l, sign_p)
24.         # Append the calculated projection to the list of projected vertices.
25.         projected.append(projected_l)
26.     # After all vertices are processed, return the boundaries of the projection.
27.     return [min(projected), max(projected)]
28.
29. # Check whether there is overlap.
30. def checkOverlap(obstacle, player, normal):
31.     # Project the player and the obstacle onto the axis given by the normal vector.

32.     obstacle_p = project(obstacle, normal)
33.     player_p = project(player, normal)
34.     # Test for overlap.
35.     if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]) or
    obstacle_p[1]-obstacle_p[0] < 1:
36.         # If the above condition is true,
37.         # it means that the projections do not overlap.
38.         return False
39.     else:
40.         # Else, it means that there is overlap.
41.         return True
42.
43. # Check for overlap and calculate projection vectors.
44. def calculateProjectionVectors(obstacle, player, normal):
45.     # Project the player and the obstacle onto the axis given by the normal vector.

46.     obstacle_p = project(obstacle, normal)
47.     player_p = project(player, normal)
48.     # Test for overlap.
49.     if (obstacle_p[1] < player_p[0]) or (obstacle_p[0] > player_p[1]) or
    obstacle_p[1]-obstacle_p[0] < 1:
50.         # If the above condition is true,
51.         # it means that the projections do not overlap.
52.         return False
53.     else:
54.         # Else, it means that there is overlap.
55.         # Calculate the values of the projection vectors.
56.         value1 = obstacle_p[0] - player_p[1]
```

```
57.         value2 = obstacle_p[1] - player_p[0]
58.         # Make them directed along the normal.
59.         vector1 = [normal[0] * value1, normal[1] * value1]
60.         vector2 = [normal[0] * value2, normal[1] * value2]
61.         # Return the necessary data.
62.         return [abs(value1), vector1, abs(value2), vector2]
63.
64. # Calculate the normal vector for a given edge.
65. def getNormal(a, b):
66.     # Obtain the vector representing the edge...
67.     edge = [b[0] - a[0], b[1] - a[1]]
68.     # ...and its length.
69.     length = math.sqrt(edge[0]**2 + edge[1]**2)
70.     # Turn the edge vector into a unit vector.
71.     edge = [edge[0] / length, edge[1] / length]
72.     # Create a vector perpendicular to the unit edge vector...
73.     normal = [-edge[1], edge[0]]
74.     # ...and return it.
75.     return normal
```

Appendix C: Main code of the game

main.py:

```
# Import the pygame library.
import pygame
import math
# Import the Separating Axis Theorem library
import sat


class ThreeDMesh():
    def __init__(self, baseColour, maxColour):
        self.z = 0
        self.baseColour = baseColour
        self.maxColour = maxColour
        self.currentColour = baseColour

    # Set the object to a given level.
    def set(self, id):
        # Reset the self.z attribute to start each level at the same z position.
        self.z = 0
        # Set the id.
        self.id = id
        # Import the data from a text file.
        self.data = self.importData()

    # This method will import polygon data from a text file.
    def importData(self):
        # Create an empty list for the data.
        data = [[[]]]
        # Create indexes that will help iterate
        # over polygons and the third dimension.
        polygonIndex = 0
        zIndex = 0
        # Open file specified by the id.
        with open('level_data/' + self.id + ".txt", 'r') as f:
            # Parse every line in the file.
            for line in f:
                if line == "#\n":
                    # If line contains a '#' it means that the data about a
                    # particular cross section has just finished. Create a new
                    # sublist for the next cross section.
                    data[zIndex].pop()
                    data.append([[]])
                    # We move to the next cross section, so we increase the zIndex.
                    zIndex += 1
                    # We have not added any polygons to this cross section yet,
                    # so we reset the polygonIndex to 0.
                    polygonIndex = 0
                elif line == "\n":
                    # If the line is emty, that means that we reached the end of
                    # a single polygon's description. We create a sublist to hold
                    # the next polygon...
                    data[zIndex].append([])
                    # ...and change the polygonIndex to indicate that we moved to
                    # the next polygon.
                    polygonIndex += 1
                else:
                    # If the line is neither empty nor it has a '#' in it,
                    # we assume that it contains coordinates of a point.
                    # We add this point to a polygon indicated by polygonIndex
                    # in the crossSection indicated by the zIndex.
```

```
60.                     data[zIndex][polygonIndex].append([float(x) for x in
    line.split(" ")])
61.             # The pop() functions throughout the code are to get rid of unpopulated
62.             # lists that occur naturally due to the construction of the data format.
63.             data.pop()
64.         # Close the file...
65.         f.closed
66.         # ...and return the processed data array.
67.         return data
68.
69.     # This method will draw a cross-section specified by self.z
70.     def draw(self, screen):
71.         # This is a set of polygons in the cross-section that
72.         # we will be drawing:
73.         drawing = self.data[math.floor(self.z)]
74.         # We iterate on the elements of the drawing list,
75.         # which are lists of vertices...
76.         for polygon in drawing:
77.             # ...and pass them to the drawing function.
78.             pygame.draw.polygon(screen, self.currentColour, polygon)
79.
80.     # This method will update the self.y attribute based on
81.     # the mouse y coordinate.
82.     def update(self, mouse_y):
83.         # Calculate the difference between the mouse position
84.         # and the current z position.
85.         diff = mouse_y - self.z
86.         # If the difference is bigger than 50,
87.         # limit the transition speed.
88.         if abs(diff) > 50:
89.             # abs(diff)/diff is used to copy the sign of diff.
90.             diff = 50 * abs(diff) / diff
91.         # Add a fraction of the difference to self.z.
92.         self.z += diff * 0.1
93.         # Set the currentColour based on self.z.
94.         self.currentColour = (calculateColour(self.baseColour[0], self.maxColour[0], se
    lf.z), calculateColour(self.baseColour[1], self.maxColour[1], self.z), calculateColour(
    self.baseColour[2], self.maxColour[2], self.z))
95.
96. # The Player class.
97. class Player():
98.     def __init__(self, x, y, width, height, baseColour, maxColour):
99.         # Set the attributes to the values given.
100.         self.x = x
101.         self.y = y
102.         self.xSpeed = 0
103.         self.ySpeed = 0
104.         self.xAcceleration = 2
105.         self.yAcceleration = 0.2
106.         self.width = width
107.         self.height = height
108.         self.baseColour = baseColour
109.         self.maxColour = maxColour
110.         self.yPV = 0
111.         # Create the vertices list
112.         self.vertices = []
113.
114.     # Update the position every refresh based on keyboard input.
115.     def update(self, xSpeed, ySpeed):
116.         # Set xSpeed based on left/right keys pressed.
117.         self.xSpeed = xSpeed * self.xAcceleration
118.         # Add a constant to the ySpeed to simulate freefall.
119.         self.ySpeed += self.yAcceleration
120.         # Set a speed limit.
121.         if self.ySpeed > 5:
122.             self.ySpeed = 5
```

```
123.         # Jump if conditions are met.
124.         if ySpeed == -1 and self.yPV < -0.1:
125.             self.ySpeed = -5
126.         # Reset the stored y component of the projection vector.
127.         self.yPV = 0
128.         # Add the speeds to the coordinates.
129.         self.x += self.xSpeed
130.         self.y += self.ySpeed
131.         # Check if the player is not slightly out of the screen on the left side.
132.         if self.x < 0:
133.             # Displace back onto the screen if yes.
134.             self.x = 0
135.         # Calculate the coordinates of the rectangle's vertices.
136.         self.vertices = [[self.x, self.y], [self.x + self.width, self.y],
    [self.x + self.width, self.y + self.height], [self.x, self.y + self.height]]
137.
138.     # Draw the Player.
139.     def draw(self, screen, levelZ):
140.         # Calculate the colour to be used while drawing.
141.         colour = (calculateColour(self.baseColour[0], self.maxColour[0], levelZ),
142.                   calculateColour(self.baseColour[1], self.maxColour[1], levelZ),
143.                   calculateColour(self.baseColour[2], self.maxColour[2], levelZ))
144.         # Use pygame's built in draw rectangle function.
145.         pygame.draw.rect(screen, colour, [self.x, self.y, self.width, self.height])
146.
147.     # Displace the player after collision.
148.     def collisionDisplace(self, projectionVector):
149.         # Change the player's x and y coordinates according to the projection vector.
150.         self.x += projectionVector[0]
151.         self.y += projectionVector[1]
152.         # Save the y component of the projection vector for use in update()
153.         self.yPV = projectionVector[1]
154.         # Reset the ySpeed after collision
155.         if projectionVector[1] < 0:
156.             self.ySpeed *= abs(projectionVector[0]) /
    math.sqrt(projectionVector[0] ** 2 + projectionVector[1] ** 2)
157.
158.     # Reset the Player's position.
159.     def reset(self):
160.         # Set the x and y coordinates to zero.
161.         self.x = 0
162.         self.y = 0
163.         # Reset the ySpeed.
164.         self.ySpeed = 0
165.
166. # The class for the lava surfaces.
167. class Lava(ThreeDMesh):
168.     # A method for detecting collisions.
169.     def collide(self, player, stars):
170.         # Take the current cross-section from the data array.
171.         cSection = self.data[math.floor(self.z)]
172.         # Iterate over the polygons in the current cross-section.
173.         for obstacle in cSection:
174.             # Check the x and y axes.
175.             if not sat.checkOverlap(obstacle, player.vertices, [1,0]):
176.                 # If there is no overlap we can jump to the next
177.                 # polygon in the data set thanks to the SAT principles.
178.                 continue
179.             if not sat.checkOverlap(obstacle, player.vertices, [0,1]):
180.                 continue
181.             # Iterate over the polygon's edges.
182.             # We assume that there is overlap unless proven otherwise.
183.             collided = 1
184.             for i in range(len(obstacle)):
185.                 # Get the normal to this edge...
186.                 normal = sat.getNormal(obstacle[i], obstacle[(i+1) % len(obstacle)])
```

```
187.                 # ...and check for overlap, if the axis is not the x or y axis.
188.                 if (normal[0]*normal[1] != 0) and
    not sat.checkOverlap(obstacle, player.vertices, normal):
189.                     # Stop checking the edges and rise the flag that
190.                     # there is no overlap.
191.                     collided = 0
192.                     break
193.             # If we got past all the overlap checks and there was overlap
194.             # on all the axes, it means that there is a collision, so we
195.             # reset the level.
196.             if collided:
197.                 player.reset()
198.                 stars.reset()
199.                 # If there is a collision we do not need to check
200.                 # the rest of the polygons.
201.                 break
202.
203. # The class for the level surfaces.
204. class Level(ThreeDMesh):
205.     # A method for detecting collisions.
206.     def collide(self, player):
207.         # Take the current cross-section from the data array.
208.         cSection = self.data[math.floor(self.z)]
209.         # The final projection vector will be a sum of all the projection
210.         # vectors from the collided polygons.
211.         finalVector = [0,0]
212.         # Iterate over the polygons in the current cross-section.
213.         for obstacle in cSection:
214.             # Create lists for holding projection vector lengths
215.             # and the vectors.
216.             projectionVectorsLenghts = []
217.             projectionVectors = []
218.             # Check the x and y axes.
219.             vectors = sat.calculateProjectionVectors(obstacle, player.vertices, [1, 0])
220.             # If the calculateProjectionVectors function did not return false,
221.             # it means that it successfully found projection vectors...
222.             if vectors:
223.                 # ...which we can add to our lists.
224.                 projectionVectorsLenghts.append(vectors[0])
225.                 projectionVectors.append(vectors[1])
226.                 projectionVectorsLenghts.append(vectors[2])
227.                 projectionVectors.append(vectors[3])
228.             else:
229.                 continue
230.
231.             vectors = sat.calculateProjectionVectors(obstacle, player.vertices, [0, 1])
232.             if vectors:
233.                 projectionVectorsLenghts.append(vectors[0])
234.                 projectionVectors.append(vectors[1])
235.                 projectionVectorsLenghts.append(vectors[2])
236.                 projectionVectors.append(vectors[3])
237.             else:
238.                 continue
239.
240.             # Iterate over the polygon's edges.
241.             # We assume that there is overlap unless proven otherwise.
242.             collided = 1
243.             for i in range(len(obstacle)):
244.                 # Get the normal to this edge...
245.                 normal = sat.getNormal(obstacle[i], obstacle[(i + 1) % len(obstacle)])
246.                 # ...and check for overlap, if the axis is not the x or y axis.
247.                 if (normal[0] * normal[1] != 0):
248.                     vectors = sat.calculateProjectionVectors(obstacle, player.vertices,
    normal)
249.                     if vectors:
250.                         projectionVectorsLenghts.append(vectors[0])
```

```
                        projectionVectors.append(vectors[1])
                        projectionVectorsLenghts.append(vectors[2])
                        projectionVectors.append(vectors[3])
                    else:
                        # Stop checking the edges and rise the flag that
                        # there is no overlap.
                        collided = 0
                        break
            # If we got past all the overlap checks and there was overlap
            # on all the axes, it means that there is a collision.
            if collided:
                # Find the index of the shortest vector...
                minimumIndex = projectionVectorsLenghts.index(
    min(projectionVectorsLenghts))
                # ...and add it to the final projection vector.
                finalVector[0] += projectionVectors[minimumIndex][0]
                finalVector[1] += projectionVectors[minimumIndex][1]
        # Pass the final projection vector to the player object.
        # If there were no collisions, it will be [0,0],
        # resulting in no displacement.
        player.collisionDisplace(finalVector)

# The class that handles all things related to stars
class Stars():
    def __init__(self, baseColour, maxColour):
        # Set the attributes
        self.z = 0
        self.baseColour = baseColour
        self.maxColour = maxColour
        self.currentColour = self.baseColour

    # Set the object to a given level.
    def set(self, id):
        # Reset the self.z attribute to start each level at the same z position.
        self.z = 0
        # Set the id.
        self.id = id
        # Reset the score.
        self.score = 0
        # Import the data from a text file
        self.data = self.importData()

    # Import the data about the stars
    def importData(self):
        # Create a temporary data list.
        data = []
        # Open the data file.
        with open('level_data/' + self.id + ".txt", 'r') as f:
            # Parse every line in the file.
            for line in f:
                # Create a temporary list for every star and append coordinates
                star = [int(x) for x in line.split(" ")]
                # Append the state of the star. 1 is uncollected.
                star.append(0)
                # Append the star list to the data list.
                data.append(star)
        # Ensure that the file is closed.
        f.closed
        # Return the data array.
        return data

    # Draw a single star.
    def drawStar(self, screen, x, y, state):
        # The basic star vertices.
        vertices = [[32,10],[20,10],[16,0],[12,10],[0,10],[9,19],[6,30],[16,24],[26,30]
    ,[23,19],[32,10]]
```

```
315.         # Add the given x and y coordinates to the star's vertices.
316.         correctedVertices = [[v[0] + x, v[1] + y] for v in vertices]
317.         # Draw the star. The state is used as width. If it is 0 (uncollected),
318.         # the polygon is filled, if it is 1 (collected),
319.         # a border of width 3 is drawn.
320.         pygame.draw.polygon(screen, self.currentColour, correctedVertices, state*3)
321.
322.     # Draw the stars and the star score.
323.     def draw(self, screen):
324.         # Iterate on the stars in the data array.
325.         for star in self.data:
326.             # Draw the stars.
327.             self.drawStar(screen, star[0], star[1], star[2])
328.         # Render the star score.
329.         for i in range(len(self.data)):
330.             if i > self.score-1:
331.                 # Draw an empty star.
332.                 self.drawStar(screen, 5+i*40, 5, 1)
333.             else:
334.                 # Draw a full star.
335.                 self.drawStar(screen, 5+i*40, 5, 0)
336.
337.     # Update the stars.
338.     def update(self, mouse_y, player):
339.         # Use the self.z and colour updating algorithm
340.         # from the ThreeDMesh class.
341.         diff = mouse_y - self.z
342.         if abs(diff) > 50:
343.             diff = 50 * abs(diff) / diff
344.         self.z += diff * 0.1
345.         self.currentColour = (
346.             calculateColour(self.baseColour[0], self.maxColour[0], self.z),
347.             calculateColour(self.baseColour[1], self.maxColour[1], self.z),
348.             calculateColour(self.baseColour[2], self.maxColour[2], self.z))
349.         # Iterate on the stars to check for collisions.
350.         for star in self.data:
351.             # Check for overlap. We get the projections by adding
352.             # width and height to respective coordinates.
353.             if not star[2] and not (player.x + player.width < star[0])\
354.                     and not (player.x > star[0] + 32)\
355.                     and not (player.y + player.height < star[1])\
356.                     and not (player.y > star[1] + 32):
357.                 # If there is a collision, set the star to collected...
358.                 star[2] = 1
359.                 # ...and add one to the score.
360.                 self.score += 1
361.
362.     # Reset the stars' state and the star score.
363.     def reset(self):
364.         # Set the score to zero.
365.         self.score = 0
366.         # Iterate on the stars...
367.         for star in self.data:
368.             # ...setting their state to uncollected.
369.             star[2] = 0
370.
371. # A class for displaying the tutorial
372. class Tutorial():
373.     def __init__(self):
374.         # Set the state and current frame to zero.
375.         self.state = 0
376.         self.frame = 0
377.         # Load the images
378.         self.firstImage = pygame.image.load("images/wsad.png").convert()
379.         self.secondImage = pygame.image.load("images/animationsheet.jpg").convert()
380.
```

```
381.     # Change to the next state
382.     def next(self):
383.         self.state += 1
384.
385.     # Draw the image corresponding to the state
386.     def draw(self, screen):
387.         if self.state == 0:
388.             # Draw the image explaining the use of the WSAD keys
389.             screen.blit(self.firstImage, [0,150])
390.         if self.state == 1:
391.             # Draw the animation explaining the concept of the third dimension.
392.             # Increase the frame counter.
393.             self.frame += 1
394.             # Calculate the frame to render.
395.             renderFrame = (self.frame//8)%24
396.             # Cut the animation sheet according to the renderFrame variable
397.             # and render it on screen. The third argument are the coordinates
398.             # and dimensions of the cut
399.             screen.blit(self.secondImage, [0,150], [0, renderFrame*200, 500, 200])
400.
401. # Calculate the colour component based on the z position.
402. def calculateColour(min, max, z):
403.     return math.floor(min + z/500 * (max-min))
404.
405. def main():
406.     # Initialize the pygame environment.
407.     pygame.init()
408.
409.     # Set the width and height of the screen.
410.     size = (500, 500)
411.     screen = pygame.display.set_mode(size)
412.
413.     # Set the title of the window.
414.     pygame.display.set_caption("My Game")
415.
416.     # This variable stores whether the user pressed
417.     # the close button on the window.
418.     done = False
419.
420.     # An object used to manage how fast the screen updates.
421.     clock = pygame.time.Clock()
422.
423.     # Creating objects for testing:
424.     level = Level((33,150,243), (13,71,161))
425.     lava = Lava((255,9,9), (180,0,0))
426.     stars = Stars((255,238,88), (253,216,53))
427.     player = Player(0, 0, 20, 20, (255,193,0), (255,111,0))
428.     tutorial = Tutorial()
429.     s = open("scores.txt", 'r')
430.     scores = [int(x) for x in s.read().split(" ")]
431.     s.close()
432.     # Load the necessary images.
433.     backgroundImage = pygame.image.load("images/main_background.png").convert()
434.     lockedImage = pygame.image.load("images/locked.png").convert_alpha()
435.     youWinImage = pygame.image.load("images/you_win.png").convert_alpha()
436.     newHighScoreImage = pygame.image.load("images/new_high_score.png").convert_alpha()

437.     prevHighScoreImage = pygame.image.load("images/prev_high_score.png").
    convert_alpha()
438.
439.     state = 0
440.     firstDraw = 1
441.     # Main program loop, runs until the close button is pressed.
442.     while not done:
443.         if state == -1:
444.             # Show the winning screen.
```

```
            if firstDraw:
                # Render the background.
                screen.blit(youWinImage, [0, 0])
                # Check if the current high score has been beaten.
                if stars.score > scores[levelIndex-1]:
                    # If yes, then draw the "New High Score" message.
                    screen.blit(newHighScoreImage, [281, 267])
                    # Change the stored high score to the current score
                    scores[levelIndex-1] = stars.score
                else:
                    # If the high score has not been beaten, render the
                    # "Current High Score" message.
                    screen.blit(prevHighScoreImage, [331, 267])
                    # Render the current high score using stars
                    # and the algorithm used for that on the main screen.
                    for i in range(3):
                        if i > scores[levelIndex-1] - 1:
                            stars.drawStar(screen, 350 + i * 33, 330, 1)
                        else:
                            stars.drawStar(screen, 350 + i * 33, 330, 0)
                # If the next level is not unlocked (and in range), unlock it.
                if levelIndex < 8 and scores[levelIndex] < 0:
                    scores[levelIndex] = 0
                # Save the scores to the scores.txt file.
                s = open("scores.txt", 'w')
                s.write(" ".join([str(x) for x in scores]))
                s.close()
                # Render the current score using stars
                # and the algorithm used for that on the main screen.
                for i in range(3):
                    if i > stars.score - 1:
                        stars.drawStar(screen, 54 + i * 33, 330, 1)
                    else:
                        stars.drawStar(screen, 54 + i * 33, 330, 0)
                # Refresh the screen
                pygame.display.flip()
                # Indicate that the screen has been drawn already.
                firstDraw = 0
            # The event loop must in this case be after the drawing part.
            # If it was not organised this way, setting firstDraw to one
            # in the event loop would trigger drawing the "You Win" screen
            # instead of the main screen.
            for event in pygame.event.get():
                # If the event type is QUIT, the user wants to close the window.
                # So we set done to True.
                if event.type == pygame.QUIT:
                    done = True
                # If any key is pressed or the mouse is clicked,
                # we go to the main screen.
                elif event.type == pygame.KEYDOWN or event.type ==
    pygame.MOUSEBUTTONDOWN:
                    firstDraw = 1
                    state = 0

        elif state == 0:
            # Show the main screen.
            # Iterate on the events given by pygame.
            for event in pygame.event.get():
                # If the event type is QUIT, the user wants to close the window.
                # So we set done to True.
                if event.type == pygame.QUIT:
                    done = True
                # If the event type is MOUSEBUTTONDOWN, we assume that the user
                # may be trying to choose a level.
                elif event.type == pygame.MOUSEBUTTONDOWN:
                    # Get the mouse position.
```

```
                    pos = pygame.mouse.get_pos()
                    mouse_x = pos[0]
                    mouse_y = pos[1]
                    # Check whether the cursor is in the level choice area.
                    if mouse_x >= 24 and mouse_x <= 475 and mouse_y >= 217 and
    mouse_y <= 438:
                        # Calculate the selected level index.
                        levelIndex = 4*((mouse_y-213)//113) + (mouse_x-24)//113 + 1
                        # If the level is unlocked, change the state.
                        if scores[levelIndex-1] >= 0:
                            # Set the data in the level-related objects.
                            level.set(str(levelIndex) + "_level")
                            lava.set(str(levelIndex) + "_lava")
                            stars.set(str(levelIndex) + "_stars")
                            # Reset the player's position...
                            player.reset()
                            # ...and all the navigation variables.
                            leftPressed = 0
                            rightPressed = 0
                            xSpeed = 0
                            ySpeed = 0
                            # Change the state to the given level.
                            state = levelIndex
            # Check if drawing needs to be done.
            if firstDraw:
                # Draw the background.
                screen.blit(backgroundImage, [0, 0])
                # Iterate on the list of scores
                for i in range(len(scores)):
                    # If the level is locked display three empty stars
                    # and a locked badge.
                    if scores[i] < 0:
                        for j in range(3):
                            # We use the drawStar() method od the Stars class.
                            stars.drawStar(screen, 31 + i%4*113 + j*33, 285 + i//4*113,
     1)
                        screen.blit(lockedImage, [28 + i%4*113, 217 + i//4*113])
                    # If the level is not locked, display the star score
                    # using a loop almost identical to that used in the draw()
                    # method of the Stars class.
                    else:
                        for j in range(3):
                            if j > scores[i] - 1:
                                stars.drawStar(screen, 31 + i%4*113 + j*33,
    285 + i//4*113, 1)
                            else:
                                stars.drawStar(screen, 31 + i%4*113 + j*33,
    285 + i//4*113, 0)
                # Refresh the screen
                pygame.display.flip()
                # Indicate that there is no need for further drawing.
                firstDraw = 0

        elif state < 9:
            # Show the level indicated by the state variable.
            # Event processing - we iterate on the events given to us by pygame:
            for event in pygame.event.get():
                # If the event type is QUIT, the user wants to close the window.
                # So we set done to True.
                if event.type == pygame.QUIT:
                    done = True
                # Handle the keydown events.
                elif event.type == pygame.KEYDOWN:
                    # Go left.
                    if event.key == pygame.K_a or event.key == pygame.K_LEFT:
                        leftPressed = 1
```

```
572.                         xSpeed = -1
573.                     # Go right.
574.                     elif event.key == pygame.K_d or event.key == pygame.K_RIGHT:
575.                         rightPressed = 1
576.                         xSpeed = 1
577.                         # Jump.
578.                     elif event.key == pygame.K_w or event.key == pygame.K_SPACE or
    event.key == pygame.K_UP:
579.                         ySpeed = -1
580.                     elif event.key == pygame.K_ESCAPE:
581.                         # If the user wants to go to the main screen,
582.                         # set firstDraw to one so that the main screen
583.                         # is drawn, and then switch state.
584.                         firstDraw = 1
585.                         state = 0
586.                 # Handle the keyup events.
587.                 elif event.type == pygame.KEYUP:
588.                     if event.key == pygame.K_a or event.key == pygame.K_LEFT:
589.                         leftPressed = 0
590.                         xSpeed = 0
591.                         # If right is still pressed, start going right.
592.                         if rightPressed == 1:
593.                             xSpeed = 1
594.                     elif event.key == pygame.K_d or event.key == pygame.K_RIGHT:
595.                         rightPressed = 0
596.                         xSpeed = 0
597.                         # If left is still pressed, start going left.
598.                         if leftPressed == 1:
599.                             xSpeed = -1
600.                     elif event.key == pygame.K_w or event.key == pygame.K_SPACE or
    event.key == pygame.K_UP:
601.                         ySpeed = 0
602.                 # Go to the next tutorial screen.
603.                 elif event.type == pygame.MOUSEBUTTONDOWN:
604.                     tutorial.next()
605.             # Get the mouse coordinates.
606.             pos = pygame.mouse.get_pos()
607.             mouse_x = pos[0]
608.             mouse_y = pos[1]
609.             # print(mouse_x, mouse_y)
610.
611.             # Game logic:
612.             # Update level and lava based on mouse position
613.             level.update(mouse_y)
614.             lava.update(mouse_y)
615.             # Move the player
616.             player.update(xSpeed, ySpeed)
617.             # Collide the player with the lava and the level
618.             lava.collide(player, stars)
619.             level.collide(player)
620.             stars.update(mouse_y, player)
621.
622.             # Do the drawing:
623.             # Set the backgorund color
624.             backgroundBaseColour = (225,245,254)
625.             backgroundMaxColour = (179,229,252)
626.             screen.fill((
627.                 calculateColour(backgroundBaseColour[0], backgroundMaxColour[0],
    level.z),
628.                 calculateColour(backgroundBaseColour[1], backgroundMaxColour[1],
    level.z),
629.                 calculateColour(backgroundBaseColour[2], backgroundMaxColour[2],
    level.z)))
630.             # Draw the lava, the level, stars and the player
631.             lava.draw(screen)
632.             level.draw(screen)
```

```
            stars.draw(screen)
            player.draw(screen, level.z)

            # Display the tutorial.
            tutorial.draw(screen)

            # Change state if player won.
            if player.x >= 500:
                firstDraw = 1
                state = -1

            # Update the screen:
            pygame.display.flip()

        # Show the frame rate in the title for performance checking.
        pygame.display.set_caption(str(clock.get_fps()))
        # Set the desired frame rate to 60fps (frames per second.
        clock.tick(60)

    # Close the window when the main loop finishes.
    pygame.quit()

if __name__ == "__main__":
    main()
```