# The Problem and the Computational Method

With the computational method already explained (click here), I can finally focus on how it relates to my problem without having to further explain what the computational method is. Firstly, the idea of producing a game digitally is no a foreign idea. While it is no easy task, all the frame work has been set up, this allowed me to focus in on how the mechanics of the game would suit the computational method rather than trying to craft a solution to the problem while applying the computational method. Think about it as taking the idea of a wheel and then making it bigger and giving it deep tires for a tractor, rather than developing an unneeded complex solution like mechanical spider legs.

I was originally uncertain about the exact design of the solution to the problem, after going through the design process, I have refined my solution to this single concept:

*"Therefore, design wise, there are six major areas for design:*

1. *Main concept/story*
2. *The playable character*
3. *The main area between levels*
4. *The physics levels**
5. *The biology levels**
6. *The chemistry levels**

**These include not only aesthetic design by movement and problem concept as well.*

*The idea of a 2D platformer was not one taken without consideration. While I will cover what I mean by a platformer and go over it with examples, I will take the time to explain my reasoning behind choosing this format for the problem in the introduction.*

*Firstly, I wanted something that would suit my target audience given the situation. The key issue was that I am expected to expose students to science that they are not familiar with. This pretty much removed any ideas of a quiz, as it would be asking too much of them. Also the idea of fun must also be upheld- my client made this **very** clear.*

*With this in mind I decided that maybe it would be better to promote an interest in science rather than make a dedicated educational resource. Stepping away from this idea of an educational resource- a promotional resource would seem to be the next step. With a younger audience in mind, it may prove more viable to pace the exposer out, interaction would also help to ensure that they are paying attention. These fields and the ever present concept of fun made me think that a short game would be the best method of approach for this task."*
*Extract from Design Introduction- An introduction of my solution.*

Originally I was split between two ideas. While they were both games, one was a 2D platformer and the other was a point-and-click style escape game. For this reason, although it has been decided that the solution will be a platformer, I will include some descriptions about the point-and-click that could have been a solution; and how it to would relate to the computational method.

Firstly, it is useful to know that almost every part of this game consists of objects. While there is also the background to consider in point-and-click and the platforms in the platformer, even those could be considered objects.

If the game itself is object orientated, then it would make sense to make using a format that is also object orientated. This reflects human thinking perfectly. If I have put my game together to consist of classes and objects, then there is no need to change or translate ideas if my paradigm also uses classes and objects (like java). This also allows the task to be more manageable by abusing inheritance between objects- if the visuals are different, but the interaction between objects are the same, I can just make them all inherit these mechanics.

This also adds options to streamline the whole process by using polymorphism, rather than having a massive directory of saved objects I can instead use a polymorphic array, allowing me to just save the object with its associated pictures and animations. This further brings out the beauty in the two types of games I have shown, each object will almost certainly fit into a polymorphic array, like reactive objects that change states when another object changes state in the platformer and interactive objects in the point-and-click that stay in your inventory.

While I have linked my solutions to the object-oriented paradigm, I have not directly linked them to the computational method. The computational method mostly focuses on breaking down the problem, then explaining the interaction between the parts. This is of course broken down into greater specifics like communication, computation coordination and design. As I have already discussed this, for the example of my actual solution, I will break the computational method into two main parts: algorithms and definition.

Object-oriented programing tackle definition by looking at a problem and seeing what it is actually made up off. If you wanted to put a book in a bookcase, object-oriented programing would define the book and the bookshelf. In would also include information about the book and the bookshelf, some may be the same (like size), while other parts of information would be completely different (like name of the book and number of rows on the shelf). Only by defining and categorising every part of the problem into classes and objects can object-oriented programing begin to work.

Instead of setting algorithms like instructions- object-oriented programing instead describes what the object will do. This is a different approach to giving commands, rather than telling and object what it needs to do in the order you want it. You instead describe the actions an object will take.

This then leaves the problem of timing. If all the objects are already defined than how will they know when to do the actions that have been assigned to them? They would need some kind of cue. I set these up as changes in states. Variables aside, most objects will either be in an off or on state. I can then set the condition for there to be a change of state depending on the class of the object:

- Interactive (the state would be changed when the user interacts with them).
- Reactive (the state will change when the state of another object changes).
- Stage (these will change in relation to each other when multiple conditions are met.)

This allows me to make algorithms based on defined objects changing state. These changes in state will open up further options for the user, as the problem that I am solving computationally is how can I keep the player away from the battery for as long as possible without denying them victory. By thinking about the problem objectively in concept, then the transition from a level designed on paper to either working or pseudo code is no long a matter a task of translating ideas, but language.

Breaking the task in aspects of a game (because that is what the result will be), I can analyse the key concepts that make up a video game and apply the computational approach to them. This is useful in this early stage of analysis, as it allows me to gain a macro view of the features of a game, without applying the somewhat more focused approach of pseudocode. When a game is decomposed, you can focus on three abstract features and apply further computational logic to each one.

The first feature would be the character. This would be represented as a single class. While the first section of the class would be concerned with self-defining logic (such as movement and asset fetching), it become possible to list all the interactions the character will have with the game and convert them into functions, while working on them like they were modules. This would allow me to slowly implement a fully functioning protagonist on the game while testing each section as I go. This iterative method is very beginner friendly as it is very forgiving on mistakes. The modular approach to the design also helps to determine what features of the character is computable. If I were to look under personal design, it would be computable to give it an assigned two-dimensional sprite by fetching existing PNG files. On the other hand, if I wanted to provide voice clips for the character, with the complexity of that task, it is not computable in this project.

The next feature would be the environment. This would consist of a collection of classes, as the environment is a collection of aspects. Unlike the robot, communications is very important for this the environment, as while consists of multiple classes and assets, it is one feature. Therefore, while not only must I design each aspect to look the part as an environment, I have to allow each part to create a whole. This requires computational aspects like; coordination between classes to allow them to blend together to create a level, prior evaluation of each class to ensure they all work together and impressive recollection between levels, as the environment is not static and will be constantly changing.

Finally, and maybe the most abstract of the three features is the story and fan interactions. This focuses heavily on the design, evaluation and coordination of the game, while stressing the importance of understanding computational abilities to create the best game with limited resources (be it time, resources or ability). The story is why people will play the game and the interactions is how the player react. While experience is definitely the best resource to have when considering these questions, inspection of pre-existing solutions to this problem and unbiased testing by random people is the logical approach, with feedback affecting the next choices, leaving to versions that may be mechanically similar, but vastly different aesthetically.

This approach of prototyping is only viable when using the computational approach. As with no physical commitment, but with a product to test (unlike the theoretical approach), I can tackle each section as a separate task, meaning that I can use different techniques and resources for each aspect of the game. This is perfect as someone who does not know how to code the whole problem at the start, as it allows all training and sources to be used with minimal consequence.