Coursework Writeup - Physics Engine - Haashim

1.0 Contents

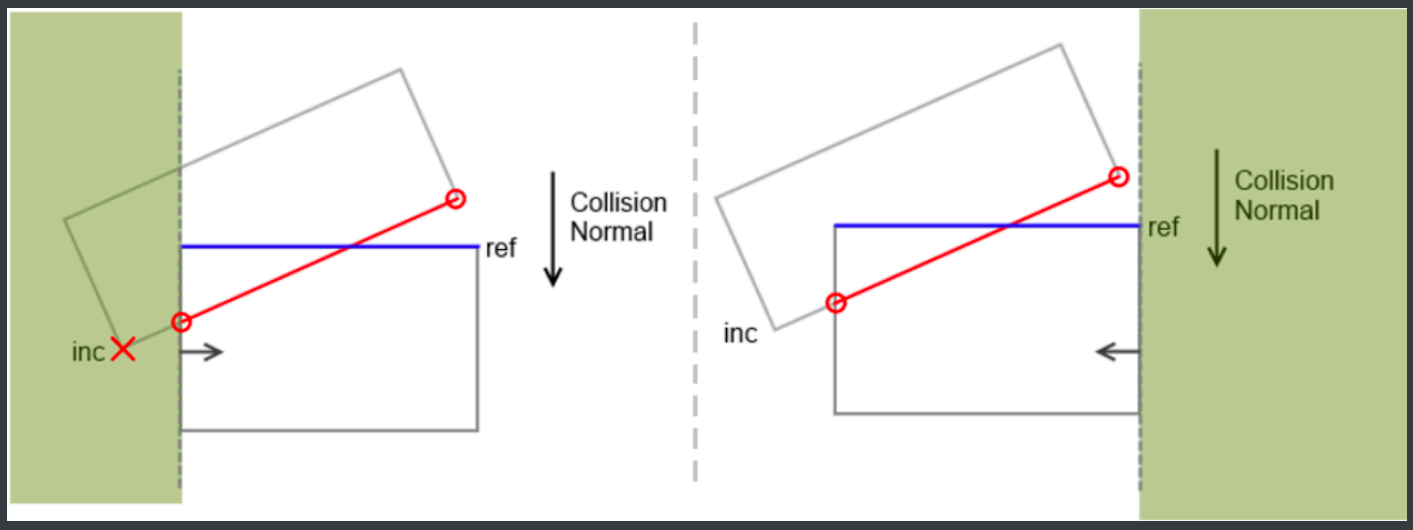
2.0 Analysis

2.1 Problem Identification:

At the moment, within a school environment, it’s impossible to model rigid bodies and their interactions. If you were to calculate trajectories and collisions manually, the process would be very time consuming and inefficient, due to the accuracy required mixed with the computation required behind it. You'd need to recompute things like acceleration and velocity and displacement frequently to give an accurate view into how the rigid bodies interact, something which would require an inordinate amount of time, if the calculus were to be done by hand, especially if it were to use corrections such as Verlet integration.

$$\overline{x}\,(t+\Delta t) = \overline{x}\; + \overline{v}\,\Delta t\; + \frac{\overline{a}\,\Delta t^2}{2} + \;\ldots \text{ so on and so forth}$$

Furthermore, the average physics engine is too resource intensive to be used in schools, and the current solutions to the problem are all too complex for usage within a school environment, usually because their target audience is programmers as opposed to teachers. Therefore, a lightweight solution that isn't too computationally intensive would be ideal. Building upon this, the problem further lends itself to a computational solution due to the aforementioned calculations. Not only is there a lot of calculus required, but there also comes the issue of collision handling and impulse resolution, which would require a contact manifold to be established too, which would mean that polygon clipping between the two incident rigid bodies would also need to take place.



All of these non-trivial calculations need to be computed on the fly and, with the aim of producing a high fidelity solution, a computer is the best suited platform on which this could be solved. In addition to these calculations however, there comes the problem of generating an output. Here, again, computers excel, since they are a great way of outputting video, and therefore it would be feasible to generate a "*live*" view of the simulation as it occurs, despite the intense frame by frame iterative calculations required.

2.2 Amenability to a Computational Solution

This problem in question lends itself to a computational solution because it involves many resource intensive calculations happening simultaneously. This leads to a very complex solution if it were to be all calculated manually, but due to the mathematical capabilities of computers they can compute tasks like these at speeds many times faster than humans. Furthermore, the problem involves calculations which are repeated many times, which means with a functional programming language the solution would be much easier to create, as opposed to doing the calculations manually, and sequentially, which would, as stated earlier, take an inordinate amount of time. In general, calculating collisions with high accuracy by hand is no trivial task, and even with a team of people it would be impossible to match the speed and accuracy of software running on a computer.

Looking more in-depth at the calculations, each frame the new displacements would need to be calculated for all rigid-bodies on the screen, using Verlet Integration. Following this, physics would need to be applied, and following this, there would need to be a collision check, initially a wide net one to identify possible collisions, followed by a smaller scale check to compute all collisions and the collision manifolds, using the Separating Axis Theorem (S.A.T) and Polygon Clipping. Finally, to resolve these collisions, I would need to apply Impulse Resolution to the Rigid Bodies to separate them, and due to all this intense calculation, a computation solution isn't just applicable; it's the only possible way it can be completed.

Another reason why the problem is suited to a computational solution is because computers can output the results of the calculations in real time. For this reason, the program would be able to generate a “live view” of the simulation, allowing for the end user to see clearly the trajectories and collisions of the Rigid Bodies involved, without being forced to just view states. When these problems are calculated manually, the bodies involved are all drawn at a fixed time, meaning the model loses its dynamic nature, and instead is only able to be represented as an image at a fixed point in time. A quick example of where this live calculation is useful is where I apply the rotations of objects onto their vertices for rendering. This would usually take a lot of time if it were done by hand, but by using a computer to apply a rotation matrix to all raw vertices of a polygon, the time taken becomes trivial.

Finally, a computational solution is best for this task because using physical demonstrations in classrooms for mechanics isn’t as accurate as simulations, because whereas in a simulation all external factors can be accounted for, in a classroom other forces such as that of the breeze could affect trajectories. Moreover, in most mathematical applications, the model is simplified so that results can be calculated, but in the real world, factors like air resistance and friction can’t be toggled on and off, leading to discrepancies between the desired interaction of Rigid Bodies and those shown via the physical demonstrations.

2.2.1 Computational Methods that the Solution is Amenable to:

2.2.1.1 Problem Identification: The problems here are quite complex but can be decomposed down into three main areas. One of these areas is handling the physics aspect, where collisions would need to be detected, and resolved, and finally external forces applied to generate both positional and angular velocities on the rigid bodies in question. Another area is the rendering, and UI, which would need to be intuitive and provide a good viewport into how the simulation is running. The final problem that needs to be solved is that of saving and loading states into the physics engine, so that models can be distributed to people or used in a game environment.

2.2.1.2 Problem Decomposition: In the case of developing a solution to this problem, it would also help to further simplify it, and break it down into smaller manageable chunks via decomposition. For each step mentioned earlier, I will break them down into smaller chunks so the reasoning is easily followed.

Firstly, for the mechanics behind the simulation:

1. Objects need to be stored in memory, as objects in an object tree.
2. Gravity and resistance need to be applied to these objects, if they are descendants of the physics frame, via Verlet Integration.
3. Collisions must be detected each frame, via S.A.T and Polygon Clipping.
4. Collisions must then be resolved each frame, via Impulse Resolution.
5. Forces must then be applied again, and the cycle should repeat until the system stabilises.

Secondly, for the UI:

1. Each frame, all objects should be rendered to the viewport.
2. The UI within the frame should be responsive throughout.
3. Inputs should be obtained from the UI and applied to the simulation.

Finally for the saving and pausing:

1. Frames should be serialised on command and output files produced.
2. The simulator should be able to load up serialised frames.
3. It should also be able to continue playing the simulation from these freeze frames.

For each of these areas, the problems and sub problems can be divided further and decomposed to produce a modular functional solution that can be efficient and defensible.

2.2.1.3 Problem Abstraction: This process will also be used in the simulation, to remove unnecessary detail from the simulation. Factors such as wind will be eliminated from the program, as well as centres of mass, because if all possible factors were to be taken into account, the problem would become incalculable in our given timeframe. Instead, objects will be abstracted and stored as particles, with vertices. By removing a dimension from 3D to 2D, the simulation will be better suited to lower power devices, which is another reason as to why abstraction here would be very useful. It will enable me to create a solution that satisfies stakeholders, and doesn't include unnecessary features that make other competitors unsuitable, and reduce the processing power required by my solution, allowing it to run well on all devices.

2.2.1.4 Backtracking will also be used in the solution, where collisions are involved, because checking each and every Rigid Body in the simulation against every other Body would be a waste of processing power and negatively impact the speed of the simulation, with a time complexity of $O(n^2)$. Instead, with backtracking, you can prune the processing and first do a wide scale search in which you check for possible collisions, and then a higher fidelity accurate search where you check these possible collisions against each other and then resolve them. In my simulation, this will be done via a Broad Phase and a Narrow Phase of collision detection. In the Broad Phase, objects will be checked with a time complexity better than $O(n^2)$, and possible collisions will be passed into a Narrow Phase detection. In my case, I decided to use the Separating Axis Theorem for my narrow

collision detection, so this will be used to determine whether or not a Broad Scale possible collision is actually colliding.

2.3 Stakeholder Identification

2.3.1 One possible client for this software would be school administrators, who could then distribute the software to departments that could use it, namely a mathematical department or a physics department. Teachers are the ideal target demographic because they will use the software professionally to explain problems and also to work through solutions in classes, and because school administrators cannot afford state of the art computers for each teacher, a lightweight solution such as the one I am proposing would be perfect for their day to day usage. An example use case of my software for them would be when they are trying to explain a new mechanics concept, for example momentum, and it's conservation, where rather than just explaining with words, they would now be able to show the class a live preview of the issue at hand. This would be much more descriptive than the formulae below which explain the same thing. They could then distribute their model to their class using the serialisation function of my software.

$$\text{Equation for the Coefficient of Restitution:}$$
$$e = \frac{v_2 - v_1}{u_1 - u_2}$$
$$\text{and the Conservation of Momentum:}$$
$$m_1 v_1 + m_2 v_2 = m_1 u_1 + m_2 u_2$$

2.3.2 Another potential stakeholder comes in the form of the students who will be exposed to the software. They will need the program to be intuitive in design, with UI that is primarily functional and then aesthetically appealing, so that they can understand what is going on without having to be taught. The program should also allow for students to download it and run it on their machines, irrespective of their own software or hardware limitations, so a high level multi-platform language like `Python` would be perfect. An example use case for students would be for them to get to know the effects of changing various parameters of the physics engine, like the coefficient of friction, restitution and masses of various rigid bodies, and then mapping out their effects in real time, something that the above formulae fail to explicitly explain intuitively.

2.3.3 A further possible stakeholder could be a developer of a game similar to Angry-Birds, where physics is integral to the gameplay, and a low power physics engine that could calculate these collisions efficiently would excel. Apps are usually run on mobile devices, which have lower processing power, so the physics engine in question would be applicable in this respect too. By including collisions between rigid bodies in my solution, it would also benefit the developer immensely since it would provide them a much richer library to build upon, and an example use case for them would be in a 2D physics based game.

2.4 Interviews

2.4.1 Questions for Physics or Mechanics teacher:

1. How do you think a physics engine would benefit your classes? *This establishes whether or not my program would be beneficial to this stakeholder*
2. Have you ever used a visualisation software like this in the past? *This would verify whether software like this has had a use-case in class previously*

3. Do you find yourself using your computer often in class? *This would act to show whether or not computers are used often in class*
4. Would you prefer a lightweight solution that can run on lots of devices? *This would highlight the importance of performance vs the capability of their device*
5. Are you satisfied with your current software when teaching without visualisation tools for mechanics models? *This would further act to assure me that there is a use case for my software*

2.4.1.1 Responses from `Teacher Interviewee`:

```
1  interview not completed
```

2.4.2 Questions for a Physics or Mechanics student:

1. How do you think a physics engine would benefit your classes? *Again, establishing utility is paramount, and I would need both the opinions of the teacher and the student*
2. Do you use a computer often in class? *This would tell me whether or not they would ever find a use case for this software*
3. As a student, do you have many visualisation tools to use? *The more tools they use, the higher chance that they would benefit from another one*
4. Do you find graphs and charts intuitive and easy to learn from? *This tells me whether or not a sample student would be predisposed to learning from a visualisation*
5. Do you think better visualisation tools would help you to learn? *Finally reinforces the utility of the software.*

2.4.2.1 Responses from `Student Interviewee`:

```
1  interview not completed
```

2.4.3 Questions for an indie game developer:

1. What current physics engines do you find yourself using today? *Identify possible competition, and a possible niche for my product*
2. Would you benefit from a lightweight solution to the problem? *Determines the utility of my solution*
3. Have you ever used Python for game development before? *My target language's utility for solving this problem would be evaluated*
4. How useful would it be to save and load states into the physics engine?

2.4.3.1 Responses from `Indie Developer Interviewee`:

```
1  interview not completed
```

2.5 Analysis of Existing Solutions

2.5.1 Box2D

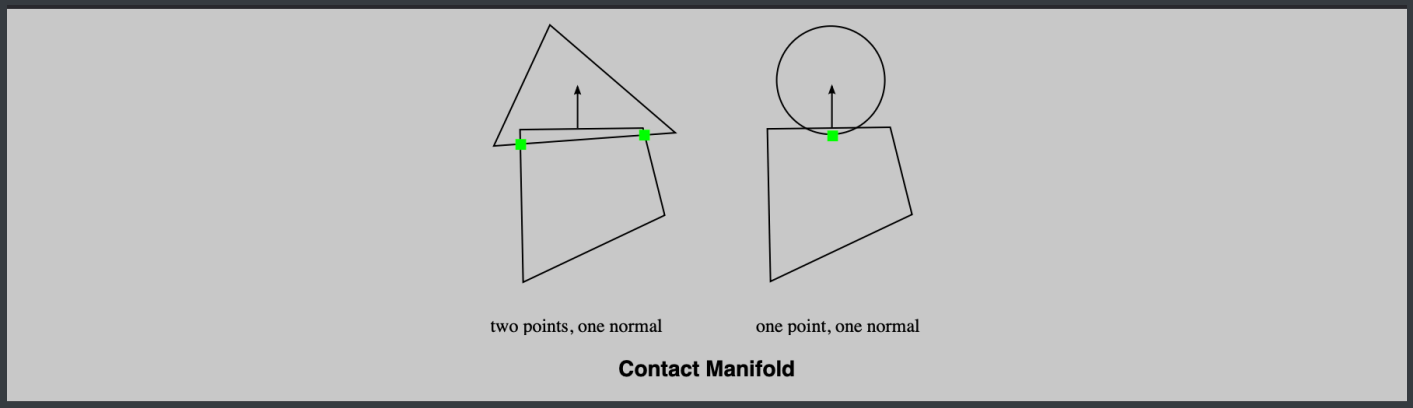
According to its creators, found on it's webpage here

> Box2D is a 2D rigid body simulation library for games. Programmers can use it in their games to make objects move in realistic ways and make the game world more interactive. From the game engine's point of view, a physics engine is just a system for procedural animation.

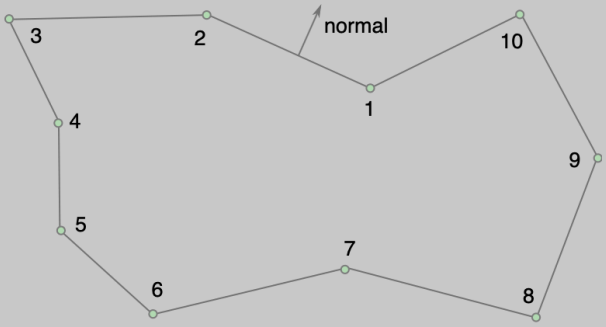
Taking a deeper look into Box2D, you can see easily that its target demographic is programmers; more specifically game designers who have experience programming in the past. Another extract from their documentation acts to solidify this idea:

> **Caution**: Box2D should not be your first `C++` project. Please learn `C++` programming, compiling, linking, and debugging before working with Box2D. There are many resources for this on the net.

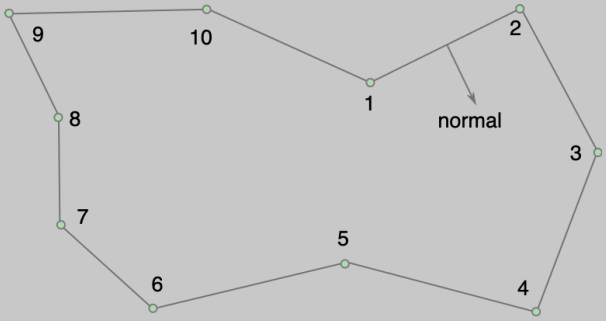
On top of this, it is written in `C++`, a compiled language, which means that it is platform specific, and if you wanted to create a piece of software that is cross platform easily, it would be quite restrictive. However, despite this platform restriction and high-level demographic which render it a quite bad teaching tool, it is much better suited for game development. Taking a deeper look into it's source, it's easy to see that this is a very robust engine, that is well suited for its target demographic.



In this first screenshot, taken again from the Box2D documentation, it is clear to see that it handles with collisions in much the same way that my engine will. It will extract, from the broad scale collisions, the narrow scale collisions, and from these successful collisions, it will then extract the contact manifold, which is a collection of vertices that represent the collision point, alongside a collision normal, which is the direction in which the shapes are colliding, and also a collision minimum translation vector, which is the depth in which one object resides in another object at a given time t. Given that Box2D is written in `C++`, it is clear to me that it will also run faster than my project, especially when it comes to certain things such as calculating collisions in frames.

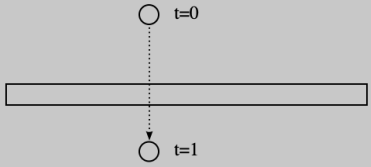


Chain Shape Outwards Loop



Chain Shape Inwards Loop

Another interesting thing that Box2D allows programmers to do is create concave shapes. Concave shapes do not work with my chosen collision algorithm, which is called S.A.T. This means that Box2D allows for a wider variety of shapes than my solution will, but I don't think that should be a problem since in teaching environments, shape concavity isn't paramount as most rigid body collisions are either between particles, which can be modelled as circles, or convex simple polygons. Box2D manages to support these complex shapes using a process called polygon vertex decomposition, which allows for the larger polygon to be broken down or decomposed into smaller convex shapes, usually triangles, for which computing collisions becomes a much more trivial task.



Tunneling

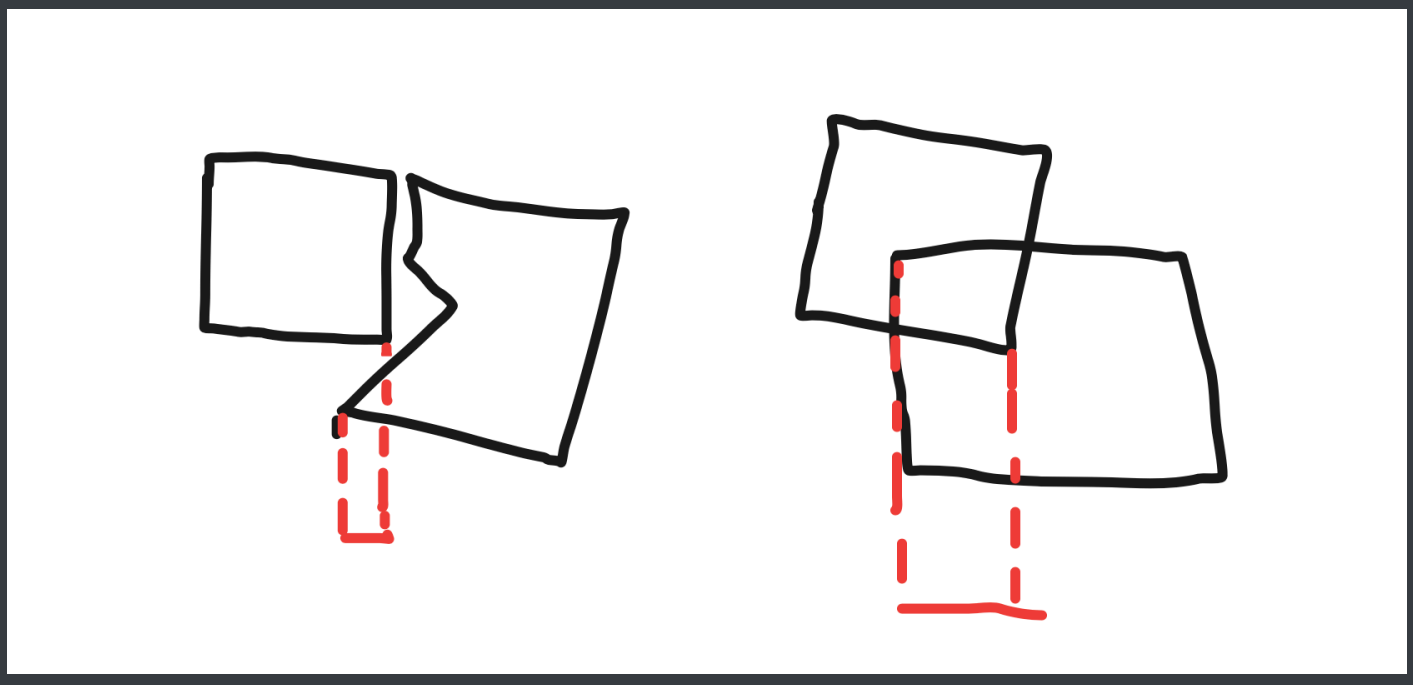
Box2D again excels in handling collisions between these complex shapes by avoiding common issues that affect physics simulations in normal scenarios. One of these is called tunnelling, which is shown above. It is where two incident shapes teleport through each other due to them moving at a high relative velocity. My solution also aims to eliminate this issue by detecting such tunnels, by detecting the movement normal of a shape and checking it against nearby shapes to see if the incident Rigid Body has clipped through. However, it is clear that Box2D, which is written in `C++` will be much faster than my `Python` based solution, at resolving issues like these.

All in all, Box2D is a very accomplished Physics Engine, and dominates this area in terms of accuracy and usage. Many big indie games such as Anrgy Birds and Happy Wheels have used this software, which goes to show its dominance in the field. However, it is a heavy application, that is processor hungry and that has a lot more features than many class teachers and students would ever need to know of or have access to. Therefore, despite it being a great solution, it's target demographic varies hugely from that of my solution, and due to this, their difference in power is understandable.

2.6 Limitations

2.6.1 Accuracy. One of the major limitations of my project is the inevitable inaccuracy of the results. Even though it will be as accurate as it can be with the calculations that I perform, which will be backed up by my utilisation of Verlet Integration, hardware limitations mean that if I want the simulation to run smoothly on all devices, despite a time corrected solution to the problem, I will need to sacrifice accuracy. Therefore, the simulation will run with only the information that it needs to know, and limit the amount of customisability of the physics solver.

2.6.2 Shapes. Another limitation of my project is, as stated earlier, the fact that it can only compute collisions between Convex Rigid Bodies, which means that any concavity with shapes will lead to unwanted interactions between the Rigid Bodies in the simulation. The reason why I cannot extend support to include concave shapes is that my collision algorithm, S.A.T, requires that shapes are convex so that when computing their projections inside each other, I don't accidentally assume two incident vertices mean that the shapes are colliding. Here is a diagram explaining the problem, with the shapes on the left being classified as colliding with my current algorithm and the ones on the right being handled correctly.



In order to solve this issue, I would need to decompose shapes into sub shapes which are convex, and then compute collisions based on these sub-shapes. However, this process of polygon decomposition is resource intensive, and due to the time frame I have when creating this solution, it wouldn't be feasible for me to also include this feature.

2.6.3 User Input. Another notable limitation of my project comes in the form of user input. This means that users are restricted to entering only a portion of the information that other solutions allow for. However, in a teaching scenario, this should be more than enough to create a realistic simulation. The different constraints that they can edit include the coefficient of restitution and friction and acceleration due to gravity, as well as the resistance to motion, masses and sizes of each of the colliding shapes. Again, as with the restriction with shape concavity, allowing for more customisation would have been detrimental to the state of the rest of the simulation due to my time frame.

2.7 Requirements

2.7.1 Software. There are a few necessary pieces of software that need to be installed on the computer before running my physics engine:

1. `Python`. Without the `Python VM`, it will be impossible to execute the completed python engine.
2. PyGame. PyGame is used for the majority of the UI in my project, which means that it needs to be installed for it to run.

2.7.2 Hardware: There are a few basic hardware limitations to my program:

1. Sufficient RAM. This will allow my program to store all the necessary objects and execute on the target computer.
2. Sufficient processing power. Without sufficient processing power, the calculations required by the physics engine wouldn't be able to be completed.

I need to research exactly what the sufficiency for both of these is.

2.8 Success Criteria

2.8.1 Programming Criteria:

1. Create a `UIBase` Class to act as a descendant class for everything else.
2. Create a `RigidBody` Class that inherits from `UIBase.`
3. Create a `UIElement` Class that inherits from `UIBase`.
4. Create an engine tree using `UIBase` objects and create a Parent-Child tree through which all objects can be accessed.
5. Create a renderer that renders the engine using recursion and prioritises certain elements over others.
6. Create a solver that solves the engine using recursion and applies physics to all `RigidBody` objects.
7. Create a `Compressor` Class that serialises and de-serialises the `RigidBody` objects in the game so that they can be saved to files and loaded.
8. Create a `Constraint` Class that allows for more control over the way objects interact with each other.

2.8.2 Physics Criteria:

1. Create a simulation that allows for `RigidBody` objects to be displayed on screen.
2. Create a simulation that allows for `RigidBody` objects to be rotated on the screen.

3. Create a simualtion that allows for `RigidBody` objects that can move and rotate across the screen.
4. Create a simulation that allows for `RigidBody` objects that experience the force of Gravity.
5. Create a simualtion that allows for `RigidBody` objects to experience collision resolution via Impulses.
6. Create a simulation that allows for `RigidBody` objects to experience correction under Verlet Integration.
7. Create a simulation that allows for conservation of momentum in collisions between `RigidBody` objects.
8. Create a simulation that allows for elasticity to be changed when `RigidBody` objects collide.
9. Create a simulation that allows for gravity to be changed.
10. Create a simulation that allows for friction to be changed when `RigidBody` objects collide, limiting lateral movement.
11. Create a simulation that allows for friction and elasticity to operate in the two opposite axes to the collision normals.

2.8.3 Collision Criteria:

1. Detect broad-scale collisions using a simple method.
2. Detect smaller-scale collisions using the S.A.T.
3. Clip polygons of the smaller-scale successful collisions using Polygon Clipping.
4. Return a contact manifold, a collision normal and a collision depth when incident `RigidBody` objects collide.
5. Resolve collisions by adding impulses to `RigidBody` objects.
6. Detect redundant collisions and anchor objects.

2.8.4 UI Criteria:

1. Create an intuitive simulation UI.
2. Allow for a Pause Button to pause the simulation.
3. Allow for a Edit Button to edit the aforementioned values.
4. Allow for a Save Button to serialise the state of the simulation.
5. Allow for a Load Button to de-serialise a simulator state.
6. Allow for a Create Button to allow for new `RigidBody` objects to be instanciated in an easy to understand way.