**Game Engine Development**

<https://github.com/Cubes58/Team_NoName_GameEngine>

As part of this project, each member of the group was assigned individual tasks to develop the Game Engine. The different sections include: Artificial Intelligence, Audio, Graphics, Level-Editor, Memory-Management and Physics. All these sections were developed accordingly and linked together using GitHub (as explained in project management).

**Project Management**

**Meetings**

To keep on top of this project, we held regular meetings to inform each member on the tasks in-hand and also aid any member and give advice on how to complete specific tasks. Below display the meetings held and what was discussed in those meetings.

|  |  |  |
| --- | --- | --- |
| Meetings Held | | |
| 08/04/2019 | 11am – 2pm | With the first meeting that was held, we split the individual tasks down into individuals based off of skill. |
| 17/04/2019 | 11am – 2pm | With this meeting, we kept each other updated in what we have done and what we would need to complete for the game engine. |
| 22/04/2019 | 11am – 2pm | We continued to update each member with the tasks in hand and also proceeded to state the tasks that needed to be complete. |
| 24/04/2019 | 11am – 2pm | To continue further work on the code and aim to have it complete by the end of the week before the next team meeting. |
| 29/05/2019 | 11am – 2pm | Complete the coding element and begin with the game itself with the models and the actual style of game we want to create. |
| 01/05/2019 | 11am – 2pm | Finalise the game element and have it completed with all the elements working. |
| 02/05/2019 | 11am – 5pm | Finalise everything in terms of the code and the game created with the engine. This would include a playthrough of the game to ensure the physics, graphics and sounds work. |

Within these meetings, various duties were modified in order to complete the game engine and the game within the timeframe. As the Final Year Project was just complete, we had a limited amount of time to complete the rest of this project. We each put in an idea on the game we wanted to create and what our tasks would be in order to complete the project. This was then worked on throughout the various weeks leading up to the deadline. As the deadline got closer, the aim was to have the individual tasks complete before we create the game itself. This was successful as we managed to create the engine with minor bugs that were fixed on the final meeting. Overall, we feel we managed the project well with our tasks and completed the project within the set timeframe. All of the individual tasks were complete with all the elements linking together successfully; with the game working and displaying what we set out.

**Code Management**

We decided to use GitHub for our project’s code management. Each section of the engine had its own branch, see Figure 1 below, which was merged every so often, and the master branch always maintained a working version of the project, which would build and run. Once we finished our section it would be merged into the master branch and that section’s branch would be deleted. The game was built using a development branch, where we could keep unfinished/broken code that was being worked on. Many commits to the GitHub repository have been made, over its lifetime.

Figure 1 - GitHub Repository Branches.

**Engine Development – Connor Rowland**

**Memory Management**

**Double Ended Stack Allocator:** A double ended stack allocator is used to allocate a massive chunk of memory, at the start of the program. This is so only a single malloc (memory request), needs to be made, over the duration of the program’s lifetime. This is done because every time memory is requested the operating system (OS) must change from user mode, to kernel mode, which is a relatively slow process (which can slow the program down, if a lot of requests are being made). A double ended stack allocator is used so it can grow inwards, from the top and from the bottom. This means static data can be allocated at one end, and dynamic data at the other, as dynamic data will change more frequently than static, and due to the nature of a stack allocator it can only clear data to a marker, which mean all data after, if at the start, or before, if at the end is wiped – certain data cannot be retained.

**Zone Allocator:** The zone allocator is given a chunk of memory, from the double ended stack allocator, so it can allocate memory for the program’s containers. This is set to the static end of the stack, because the zone allocator shouldn’t grow/shrink itself, but the contents (different, dynamic, nodes) within it can. The zone allocator implements a doubly linked list, to manage the different memory blocks allocated for the containers. At the start there’s a single node, which is identified as a free node (unoccupied), this encapsulates all the free bytes, given by the stack allocator. Once a container is created, and it requests data this free node is split into two nodes - an occupied node, and an unoccupied node. The occupied node will have enough space for the node’s header information, and for the data block requested, by the container, to fit the full capacity of elements in. The unoccupied node will shrink and contain the remaining free bytes. The next/previous pointers are amended, so the new node’s next pointer points to the old node, and the old node’s previous pointer points to the new node.

When a node is freed the zone allocator checks whether it can merge with its next or previous zone nodes, so it can create a bigger free zone node (chunk of memory), this should reduce fragmentation, as it allows free memory space to merge, and can split into smaller nodes, if requested to, and then re-merge again, once the memory is no longer occupied.

**Containers:** They’re two types of containers, that take advantage of the zone allocator, these include: The vector, and the Hash Map.

**Vector:** The vector allocates a chunk of memory, for a single type of elements, enough memory for the maximum capacity, and elements can be added/removed. Once an element is removed the index is added to a list, which keeps track of gaps in-between the data, then when an element is pushed it’ll check, to see if there’s any gaps, if there is a gap that element will be added to the gap (removing it), otherwise it’ll be added to the end. The vector can grow; when an object is pushed, and cannot fit within the vector’s current capacity, it’ll grow by the vector growth scalar (set to 2), and the pushed element will be added. All the content of the vector must be copied to the new zone node, which invalidates every pointer that points to the elements within the vector.

**Hash Map:** Like the vector, the HashMap requests a chunk of memory, (enough) for the maximum capacity of elements it can hold. The difference is how the memory is managed (added/deleted/searched for). The data is still stored in a contiguous block of memory, but the node’s point to each other, creating a tree design, which makes searching for some data quicker. Like the vector when the hash-map grows beyond its capacity it’ll request a new (bigger) zone node, copy its current content over, and add the new element in.

**Memory Manager:** The singleton design pattern is used, to create a memory manager instance, which initializes, manages and cleans up the memory. It created the stack allocator and zone allocator. The containers include this to request memory. Memory can be requested from the stack, or from the zone allocator, and then freed.

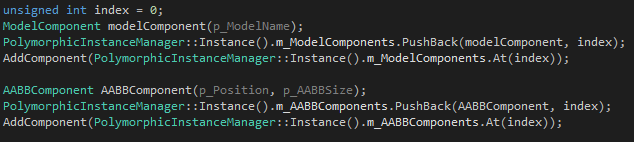
**Public Interface:** The Memory Manager’s static instance is used to request/free memory. The custom containers (vector and hash-map) also request memory from the memory manager.

Requesting memory from the stack returns a MemoryBlock object, which is allocated on automatic memory, this stores a pointer to the start of the memory block, a pointer to the end, and an unsigned integer, which stores the size, of the memory block. To free memory from the stack provide the memory manager (singleton) instance with a marker (unsigned char pointer), and it will delete everything between the provided marker and the end of the section marker.

Requesting memory from the zone allocator returns a pointer to a ZoneNode instance. The zone node instance and the zone node’s free data block are tied together, within the dynamic/heap memory (header information, about the zone node, and then the free memory block starts after the header information, the header size is 32/64 bytes: 4 unsigned char pointers, unsigned int, Boolean, and two zone node pointers). To free memory from the zone allocator a pointer to the zone node that’s being freed is passed, once this is passed the memory will be freed, and maybe the zone node will merge with surrounding zone nodes.

The Memory Manager also has a clear method, which deletes/cleans up the entire block of allocated memory (allocated by the double ended stack allocator), this should only be used if the memory manager is no longer needed.

Before the memory management code was implemented, for the game components game, the game had performance issue when compiled in Debug mode. The performance difference surprised me, especially considering I implemented it for the game object components. I assume this is due to all the cache misses caused by the object-oriented component design pattern, and the components being “randomly” located in the memory – heavy use of get component. See the code below, for an example of how this is done.



**Graphics Engine – Thomas Masdin**

**Aim**: Separate the graphics side of the game engine into its own engine which handles the all the rendering and graphics and it slots in to the Scene class to get game objects, can render text on command and allows for ease of use when it comes to post processing effects.

**Main Render Engine**: This handles the entire rendering side; it sets up all the shaders that is used in the system and allows them to be edited outside of the engine at runtime through the use of getters and setters. It passes all the necessary information that most shaders should need so it does require some structure to how the shaders are laid out.

**Framebuffer Object**: Class that can be used to create multiple different types of framebuffers used throughout the engine (e.g. postprocessing, shadows), it uses enum indicator FrameBufferType to decide what framebuffer needs to be created to optimise it for its indicated purpose as opposed to adjusting/recreating an individual framebuffer.

**Post Processing:** Class that holds all the screen space post processing effects which is called after the main render loop (Requires the main render buffer colour texture) and passes them to each called post processing effect.

**Shadows:** Class that handles the shadow mapping (Directional and Point) for the entire scene, it utilises the above framebuffer object class and renders the scene to a depth buffer using a simple depth shader (& geometry shader for point shadows).

**Lights**: Simple class that is used to create a light for the scene, this is then added to a list and the necessary information is passed to the shaders/shadow class (Has debug render mode which creates a cube around the lights position)

**Shader Program**: Basic shader program class that simplifies passing information to the shader needing only the data type and the uniform name, it also handles error checking by flagging when an error has occurred in the program as well as console printing (this can be adjusted using the font renderer).

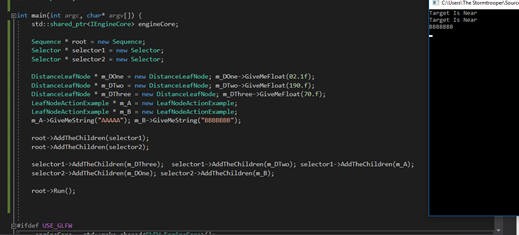
**Font Renderer:** Automates the font rendering side of the engine with a simple font shader (Can change font style at runtime) which renders the font to screen space quads. For rendering specific text, it only requires the position, size and what text to render and it automatically renders it to the screen when called.

**Artificial Intelligence – Edward Magee**

**Behavioural Tree:** The way this works is you create a root which is the sequence, this will be inside each NPC or object that needs multiple states. Then you push selectors into that root each selector will contain leaf nodes, they will either be a conditional which can return a true or false depending on the value passed in. If it becomes true you can make the next one another conditional or an action, if its false it won’t reach the next node.

Each object will have this inside of them, values are passed into it based to check these conditions and based on those conditions certain actions will be activated.  Because of this more leaf classes will have to be added later, further into development more and more can be added however they can also be re used if another NPC’s or object need them to something similar making it reusable.

**Behavioural Tree (Testing):** To test the behaviour tree, I created a class which takes in a string and made this the action node for debugging it. I had two selectors the first one took in two conditional nodes each with different and the other one only took in one. If the value is above hundred it should not do the action and move onto the next selector. In this test you can see that it does that as m\_Dtwo is over the values and is not printing to the console where as m\_Dthree is.



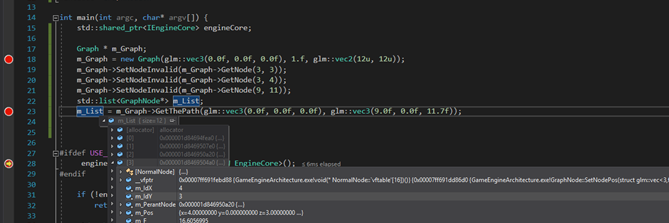
**Pathfinding (Nodes):** Each node will be a position in the map that stores information for the NPC allowing them to make better position. Each node needs to contain information, ID, the node it came from (Parent Node), its position. It also needs to contain information to help with this pathfinding like G, H and F. G stores the distance from this node to the starting node, H is the same but from the end node and F is these totals added together. IsDiagonal is also used for pathfinding to see if the node is in the top left, bottom right etc.

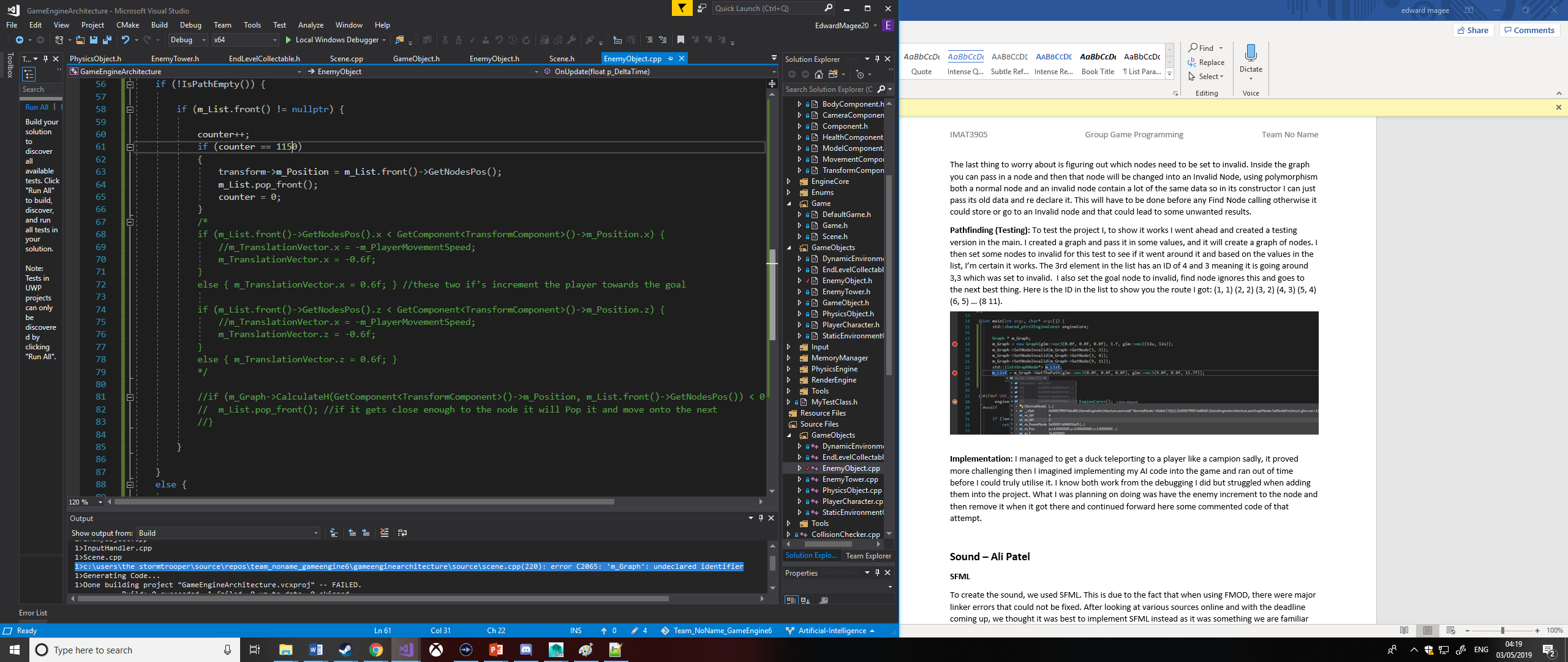
**Pathfinding (Graph):** The graph contains a 2D vector, containing the nodes previously discussed I used the find node method to find the closet node to that position, once it finds it will then then return that node. It expects a vec 3 as a starting point to build the graph, float distance is how far each node is between each other and the vec 2 shows how many nodes are going to be in this graph.

Get the path method is pretty much the A star as I’m writing this takes in two vec 3’s and then finds the start and end node that way. Sets everything to 0 or null so lingering values do not get in the way of the current search. While the open list is empty it will be checking nodes and the surrounding nodes. Using the get children method I get the surrounding nodes I then loop through these children to check if they are valid. Later, in the loop the G value (which is distance from start), H value (distance to target) are calculated. I calculate the H value by using Pythagoras, I know the goal and start so I just use the hypotenuse as the distance. The parent node is also set so it remembers where it came from. Once the goal node is found it will stop the looping and construct the path, each node stores the previous node so I can loop and create a list and then that list will become the path.

When using this code you would start by constructing it, and each enemy will have a position so that can be passed into it, it will return a list and every time it gets to a node (each node has world position stored in it) it will pop the front element and move onto the next. One improvement is instead of using Find Node each time you would pass in a node and have the object find its nearest node at the start.

The last thing to worry about is figuring out which nodes need to be set to invalid. Inside the graph you can pass in a node and then that node will be changed into an Invalid Node, using polymorphism both a normal node and an invalid node contain a lot of the same data so in its constructor I can just pass its old data and re declare it. This will have to be done before any Find Node calling otherwise it could store or go to an Invalid node and that could lead to some unwanted results.

**Pathfinding (Testing):** To test the project I, to show it works I went ahead and created a testing version in the main. I created a graph and pass it in some values, and it will create a graph of nodes. I then set some nodes to invalid for this test to see if it went around it and based on the values in the list, I’m certain it works. The 3rd element in the list has an ID of 4 and 3 meaning it is going around 3,3 which was set to invalid.  I also set the goal node to invalid, find node ignores this and goes to the next best thing. Here is the ID in the list to show you the route I got: (1, 1) (2, 2) (3, 2) (4, 3) (5, 4) (6, 5) … (8 11).

**Implementation:** I managed to get a duck teleporting to a player like a companion, sadly, it proved more challenging then I imagined implementing my AI code into the game and ran out of time before I could truly utilise it. I know both work from the debugging (shown in the images above) but I struggled when adding them into the project. What I was planning on doing was have the enemy increment to the node and then remove it when it got there and continued forward here some commented code of that attempt. In the image below is the was the code I tried to get working it would move the object towards the goal and then pop it when it arrived. When the pathfinding was completed, I would have liked to add the behaviour tree to wander (if the player is too far away using the h cost method) or charge at the player and reset their position etc.

**Sound – Ali Patel**

**SFML**

To create the sound, we used SFML. This is due to the fact that when using FMOD, there were major linker errors that could not be fixed. After looking at various sources online and with the deadline coming up, we thought it was best to implement SFML instead as it was something we are familiar with and would be just as good.

**Audio Manager**

An audio manager was created to get the sound working with the game. What I wanted to achieve was for a specific audio to play when an action has been complete within the game. In this case, when the duck moves, a quacking sound would play and whenever it would reach the flag, a winning sound would play to signify it. At first, there was a sound buffer which stores the sound effects to be played. There was another object called sound that is used to play the sound buffer. The sound effects store the sound buffer and the objects in which they are bound to. The name of the sound effect is in its key. ‘m\_BackgroundMusic’ manages the background music. The audio manager is singleton class. This prevents unnecessary duplication of assets, in this case sounds.

With the audio engine, it would begin streaming an audio file and checks whether the sound/music should be playing. It would then return true if the audio file can be streamed otherwise false. When a sound is loaded from a sound buffer, it would store it into the memory. This is due to the file being small enough to be stored whereas the music file is unable due to the size. A sound effect would then be requested to ensure that it can be played at the correct time. The duck sounds would be played at random intervals, so it would seem realistic when playing the game.

With the Audio Manager, it would work as a separate Visual Studio program but when implementing it with the game engine, it would throw up an exception error. This error was something that could not be fixed, and the issue could not be found. All the settings matched up with the game engine however, the game engine would display errors. The other program that was created to test sound, displays the sound working correctly. After numerous searches on the web and asking peers, it still did not work.

**Physics Engine: Sam Knight**

**Game Loop:** When designing a game loop, certain factors need to be considered, such as: different computers have varying processing power, and this should not result in a potentially unfair advantages when the game is played on a faster or slower machine. We also want to prevent large fluctuations in the elapsed time of a frame, since this can result in object phasing through floors/walls.

The core to a good game loop is to pass a constant amount of time to each update method, once that amount of time has elapsed, rather than just using the elapsed time per-frame. In other words, we only tell the game to update every… let’s say 0.016 seconds, this is called the frame period and it never changes - we will call this update a physics frame. To keep track of the time over several frames we accumulate the elapsed time per frame, once this exceeds the physics frame period of 0.016s we do a physics frame. finally, to reset we subtract the frame period from the accumulated time, this will usually result in a small amount of time left over, this results in a balancing act that keeps the physics frame happening at accurate times even if there is suddenly a large amount of calculations that slow the game down. If the game is running smoothly the physics frames will happen every 0.016 seconds and if it slows down the game might do 2 physics frames in quick succession to compensate for the lost time.

As much as we want to make the game fair, we also want the game to look as smooth as possible regardless of the machine - if you have high-specs the you should, visually, get your money's worth. Therefore, we do render every frame, keeping it as fast as possible as separate from the physics frames.

**Collision Detection:** There are a variety of collisions to detect depending on the circumstances: we could have two spheres, a sphere and plane, a sphere and axis aligned bounding box or two AABBs. All of these require their own check methods. Most of these checks are relatively simple: point tests require calculating distances from another object e.g. a plane, sphere or AABB; sphere tests calculate the distance and subtract the radii; and AABB tests do several axis corner tests.

Every physics frame we iterate through all the game objects and check them for collision components i.e. AABB component, if these components exits, we can check this object against all other objects that have a collision component. If the collision checker returns true, then we can move onto resolving this collision. To resolve collision, we use position data we recorded before any movement happened and revert the object to that, therefore if we have a dynamic object falling due to gravity and a static floor (both with AABB components), the falling object will stop when reaching the floor.

**Dynamic/Kinematic/Static Body:** To create realistic motion we need a component that can simulate this motion based on given factors such as velocity, acceleration, mass, restitution, drag and friction. We also need a criteria for what type of motion we want the object to provide: we might want the object to be unmovable regardless of the forces, acceleration and velocities applied to it, this is the static type; we might want the object to work with constants e.g. the acceleration stays as what it is set as, this would be useful for objects like a moving platform; finally, the dynamic object type means it is the object will only experience a force/acceleration as long as a force/acceleration is applied to an object (every physics frame the dynamic body’s acceleration is set to 0), therefore, it only experiences forces for as long as there applied.

**Quad Tree:** We considered using a horizontal quad tree to reduce the amount of unnecessary collision checks. However, during the implementation, it became obvious that this would have flaws. Say we have two adjacent quadrants, and two AABB, one in each quadrant, if both AABBs were overlapping at the edge of the quadrants, then no collision would be detected. This is the result of the quad tree only checking collisions for objects in the *same* quadrant. The fix to this wasn't immediately obvious since it isn't a simple case of also grabbing everything in an adjacent quadrant since that adjacent quadrant could be split into many quadrants depending on the number of objects in it. To improve this project, I would take a second look at how to implement a feature like this effectively, since it would be a great performance increase for scenes with many objects.

**Level Editor: Aaditya Pancholi**

ImGui is an immediate mode graphical user interface library. This library makes a graphical interface for all sorts of software’s.

When creating the ImGui for the game engine, I first had to research how ImGui is implemented within an engine. To do this, I downloaded the files from the main ImGui GitHub repository. I then followed an example of which files to use within our own game engine. As we are using OpenGL and using the glfw interface, I copied the appropriate files into our engine. The main issue that I came across is linking the files and editing each file to make it work with our engine. To do this, I had to copy the files into our engine directory and then create a render method in the engine render file. At first, I created it within the scene but had issues linking the transform component to the scene file. To fix this I then copied the code from the scene into the render engine file as all the Transform Components are initialised there. The ImGui UI started to show up in the window meaning that the linking error was fixed. I then moved on to creating additional dropdown menus and adding in a scaling on the x, y and z axis.

**Conclusion**

In conclusion, as a group, we feel we worked well together. Our knowledge and strengths have been enhanced while working on this project. The game engine works successfully while managing to create a working game with it. There are various sections that can be improved, but within this timeframe, we feel this game engine created performs to how we would like it to.

We also feel, the way we managed the project was successful as having regular meetings and using GitHub enabled all members to keep on top of the work and keep on track to complete within the timeframe.

**Team Members:**

* Connor Rowland (P16204361) – Memory Management
* Thomas Masdin (P16189498) – Graphics
* Sam Knight (P16184152) – Physics
* Edward Magee (P16188123) – Artificial Intelligence
* Ali Patel (P16199446) – Sound
* Aaditya Pancholi (P16198363) – Level-Editor