Main				
Identifier	**Parameters**	**Return**	**Access**	**Description**
main()	void	Int	Initial Call	Base function in program, creates initial scenario objects and dispatches the simulation thread, as well as initialising the GLFW library.
windowSetup()	void	GLFWwindow*	Public / main.cpp	Contains boilerplate GLFW code to initialize a window that can has an OpenGL context, will exit if GLFW or OpenGL cannot initialise correctly.
initDisplay()	int lXRes, int lYRes	void	Public / main.cpp	Called to set-up the initial OpenGL perspective and orthographic matrices to correctly map the OpenGL view-port and rendering to the size of the window (Passed parameters)
setupDefaultScenario()	render* renderAP, shared* sharedAP	void	Public / main.cpp	Sets up a default scenario into the render scenario storage using the render object access pointer, it will then also update the shared area with the scenario in render.
startup()	shared* sharedAP	void	Public / main.cpp	This function is passed to the threading component in the C++11 standard library and serves as the 'main' for the secondary simulation thread, it will execute concurrently with the main thread. To begin with, it creates its own local storage using the simulation object, it also creates a mutex (Mutual Exclusion) object which is used to make the thread pause and wait to be unlocked by another thread, allowing synchronisation of the threads. The simulation loop will update its own control storage from the shared area, and will perform iterations and send simulation updates if it is running and get updates from shared if it is paused. If an exit is called, the function will unset the shared areas exit to notify the main thread that it has exited.

Body (Class)				
Identifier	**Parameters**	**Return**	**Access**	**Description**
body()	void	void	Public / body	Creates a body object leaving all values to their default. (0)
body()	body* p_b	void	Public / body	Creates a body object copying all attributes from the pointer to a body object that is passed.
body()	double p_m, double p_r, double p_pX, double p_pY, double p_pY, bool p_fixed	void	Public / body	Creates a body object, allowing setting of attributes for mass, radius, position, and if the body is fixed or not.
body()	double p_m, double p_r, double p_pX, double p_pY, double p_pY, double p_vX, double vY	void	Public / body	Creates a body object, allowing setting of attributes for mass, radius, position, velocity.
body()	double p_m, double p_r, double p_pX, double p_pY, double p_pY, double p_vX, double vY, float p_color[3]	void	Public / body	Creates a body object, allowing setting of attributes for mass, radius, position, velocity and color.
~body()	void	void	Public / body	Called on the destruction of the body object. Implementation is empty
calcPosition()	double p_dt	void	Public / body	Calculates and populates the values for X and Y position based on the current position, velocity of the object and the delta time of the simulation.
calcHalfVelocity()	double p_dt	void	Public / body	Calculates and populates the values for X and Y velocity based on the current velocity, acceleration of the object and the delta time of the simulation.
calcMomentum()	int xy	double	Public / body	Calculates and returns the current momentum of the body object in the component designated by xy. p=mv

Scenario (Class)				
Identifier	**Parameters**	**Return**	**Access**	**Description**
scenario()	void	N/A	Public / scenario	Constructor, used for memory allocation, sets default values in the object control structure
virtual ~scenario()	void	N/A	Public / scenario	Virtual destructor, called on object destruction, calls deleteAllBodies() to correctly free memory, virtual to allow use as a base class in inheritance structure.
addBody()	body* p_nb	void	Public / scenario	Takes in a pointer to a body object and will add it onto the end of the current body vector.
delBody()	int p_bdID	void	Public / scenario	Deletes a body at the vector position (p_bdID) and frees the memory associated with that particular object.
deleteAllBodies()	void	void	Public / scenario	Deletes all bodies in the storage vector.
updateBody()	body* p_nb, int bodyID	void	Public / scenario	Updates a body (bodyID) in the scenario storage vector with the information in the body object pointer. (p_nb)
updateBodies()	std::vector<body*> p_bodies	void	Public / scenario	Updates the current body store with the one passed, it calls deleteAllBodies to free all memory used by the current body store.
updateControl()	control p_control	void	Public / scenario	Updates the control store with the one passed as a parameter.
getBodies()	void	std::vector<body*>	Public / scenario	Returns the current body store in the object, pointers are the same.
getControl()	void	control	Public / scenario	Returns the current control store in the object.
getPaused()	void	bool	Public / scenario	Returns the current boolean value of paused in control structure.
getExit()	void	bool	Public / scenario	Returns the current boolean value of exit in control structure.
setPaused()	bool tf	void	Public / scenario	Sets the boolean paused value in the control structure.
setExit()	bool tf	void	Public / scenario	Sets the boolean exit value in the control structure.

Render (Class)				
Identifier	**Parameters**	**Return**	**Access**	**Description**
drawBody()	body* p_b	void	Private / render	This function takes in a pointer to a body object and will render a body at the position of that object in world space, the circle drawing makes use of the radius of the body to give it a variable size.
createSuperstructure()	int p_soBodies, double p_cMass, double p_oMass, double p_cRadius, double p_oRadius, double p_cPosX, double p_cPosY, double p_cVelX, double p_cVelY, double p_coSpacing, double p_sRadius, float p_color[3]	void	Private / render	This function makes takes in a large number of variables in order to generate a complex structure using pseudorandom number generation, confining the points to a circle, it also provides control for the spacing from the center point and the radius of the system. For each body the velocity is calculated that allows it to orbit the central body in a circular orbit.
drawScene()	void	void	Private / render	A for loop that will render every single body in the current body store.
checkCoord()	double x, double y, double ad	int	Private / render	Check every single body against the coordinates provided and the extra radius around that point, returns the ID of the body at that position or -1 if no bodies are present.

Simulation (Class)				
Identifier	**Parameters**	**Return**	**Access**	**Description**
getComponentDistance()	body* bA, body* bB, int xy	double	Private / simulation	Returns the component (XY) distance bettween two bodies.
getVectorDistance()	double dX, double dY	double	Private / simulation	Returns the total distance using the X and Y distances provided, uses pythagoras theorem. a^2 + b^2 = c^2
resetAllAcceleration()	void	void	Private / simulation	Resets the acceleration variables in every body to zero to allow for correct summation of acceleration.
calcAcceleration()	body* bA, body* bB	void	Private / simulation	Calculates the acceleration acting on a pair of bodies due to the gravitational constant, calculating the force initially and dividing by the individual bodies mass to get acceleration. F=(GMm)/r^3 x r(xy) and F=ma
calcAllAcceleration()	void	void	Private / simulation	Calculate all accelerations for every body in the simulation, uses a 'half-matrix' double for loop traversal as each calcAcceleration() call does both bodies in the relationship.
calcAllCollisions()	void	void	Private / simulation	Checks for and calculates the result of any collisions that may occur, position is a weighted mean based on mass, velocity is calculated through momentum, radius is converted to area and summed, colours are averaged.
calcAllHalfVelocity()	void	void	Private / simulation	Calculates half iteration velocity for every body in the current store based on acceleration and delta time.
calcAllPosition()	void	void	Private / simulation	Calculates the new position for very body based on current velocity and delta time.
initialCalc()	void	void	Public / simulation	Calculates the initial acceleration and checks collisions, called at initial start of the simulation to give initial acceleration values and to remove bodies that are too close together.
iteration()	void	void	Public / simulation	Simulates the scenario for a single iteration, using leapfrog iteration with synchronised calculations, 1/2v -> r -> a -> 1/2v
getIPF()	void	int	Public / simulation	Returns the current value of IPF in control structure, used by sim loop.

Shared (Class)				
Identifier	**Parameters**	**Return**	**Access**	**Description**
updateBodies()	std::vector<body*> p_bodies	void	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
updateControl()	control p_control	void	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
getBodies()	void	std::vector<body*>	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
getContriol()	void	control	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
getPaused()	void	bool	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
getExit()	void	bool	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
setPaused()	bool tf	void	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)
setExit()	bool tf	void	Public / shared	Performs the same operation as the function in the scenario base class, but overrides this function to include a mutex lock to prevent concurrent read write access. (Polymorphic)

UI				
Identifier	**Parameters**	**Return**	**Access**	**Description**
applyCamera()	void	void	Public / ui.hpp	Applys zoom (scale) and move (transform) operations to the OpenGL GL_MODELVIEW matrix based on the global variables in the ui.cpp file, which are modified by callbacks.
getMouseHeld()	GLFWwindow* window, int button	bool	Private / ui.cpp	This function uses static variables that are retained even once the function goes out of scope, using GLFW timer, the function checks that the mouse button passed has been held for at least 0.15 seconds before returning true, polled.
moveCamera()	GLFWwindow* window, double cursorX, double cursorY	void	Private / ui.cpp	Checks that the mouse is held down using getMouseHeld() and works out the difference bettween the last set of mouse coordninates stored in static variables, providing a transformation vector by which is used by applyCamera(). (Global)
zoomCamera()	double change	void	Private / ui.cpp	Changes the scale factor by the parameter passed, multiplied by changeFactor, which is dependant on the current scaleFactor / 5, this ensures that zooming is responsive at all zoom levels. (Feels consistent)
getCoord()	GLFWwindow* window, double &aX, double &aY	void	Private / ui.cpp	This function gets the current mouse cursor position through parameter window, using inbuilt function gluUnProject() to convert the mouse coordinates to world space coordinate. (Gets the current Projection, ModelView and Viewport matricies)
cursorPosCallback()	GLFWwindow* window, int cursorX, int cursorY	void	Private / ui.cpp	This function is called by GLFW when the cursor position changes inside the window, this also contains the input passthrough to the AntTweakBar library, if ATB does not handle the input, moveCamera() is called.
mouseButtonCallback()	GLFWwindow* window, int button, int action, int mods	void	Private / ui.cpp	This function is called by GLFW when there is a mouse button pressed or released, it is also passed to ATB, this function uses getCoord to get the cursor world coordinates and then uses checkCoord in the global render object pointer to check if a body is present at that location, if there is, that body becomes the activeID and updateUI is called.
mouseScrollCallback()	GLFWwindow* window, double xoffset, double yoffset	void	Private / ui.cpp	This function is called by GLFW when there is a mouse scroll event, it is also passed to ATB, this function passes the yOffset to zoomCamera() if ATB does not handle the input.
keyboardKeyCallback()	GLFWwindow* window, int key, int scancode, int action, int mods	void	Private / ui.cpp	This function is called by GLFW when there is a keyboard key pressed, it is only used by ATB in this case.
keyboardCharCallback()	GLFWwindow* window, int codepoint	void	Private / ui.cpp	This function is called by GLFW when there is a keyboard key pressed, it is only used by ATB in this case.
windowResizeCallback()	GLFWwindow* window, int width, int height	void	Private / ui.cpp	This function is called by GLFW when the window is resized, it is similar in function to initDisplay() in main, redefining the projection and viewport based on the window size and then applying the camera again to scale and transform.
setCallbacks()	GLFWwindow* window	void	Public / ui.cpp	This function is called at the start of the program and assigns the previous GLFW callback functions so that they are called when their respective events occur.
updateUI()	render* renderAP	void	Public / ui.cpp	This function deletes the body object pointed to by activeBody, updates current body count variable and creates a copy of the object at the pointer in the body store at activeID. If no bodies are present then a null body object is created, this has no relation to the body storage.

UI				
Identifier	**Parameters**	**Return**	**Access**	**Description**
updateUI()	render* renderAP	void	Public / ui.cpp	This function deletes the body object pointed to by activeBody, updates current body count variable and creates a copy of the object at the pointer in the body store at activeID. If no bodies are present then a null body object is created, this has no relation to the body storage.
updateBody()	render* renderAP	void	Public / ui.cpp	This function will update the currently selected body in the body storage (render) with the current activeBody and activeID, does nothing if bodyCount = 0;
deleteBodyButton()	void *cData	void	Private / ui.cpp	This function will delete the current selected body, it is a callback from an AntTweakBar button. It is passed a void pointer, meaning that it can point to any data type and then recast back to the actual data type, in this case it is used to pass a pointer to the Render object. Assuming the body count is not 0, it will delete the current selected body from body storage and then call updateUI.
deleteAllBodiesButton()	void *cData	void	Private / ui.cpp	This function will delete all bodies in the scenario, using the same method to gain access to the Render object. Calls deleteAllBodies().
newBodyButton()	void *cData	void	Private / ui.cpp	This function will create a new Body in the scenario, starting at 0, 0 with mass 1 and radius 1 and with no velocity, using the same method to gain access to the Render object.
newSuperStructureButton()	void *cData	void	Private / ui.cpp	This function will create a new Superstructure in the scenario, based on values that are currently set in the superstructure data structure, defined by the superstructure UI. Uses the same method to gain access to the Render object.
setupSimGUI()	render* renderAP	void	Private / ui.cpp	This function sets up all the variables that are accessed in the simulation control GUI, as well the window properties.
setupBodyGUI()	render* renderAP	void	Private / ui.cpp	This function sets up all the variables that are accessed in the body control GUI, as well the window properties.
setupSuperStructGUI()	render* renderAP	void	Private / ui.cpp	This function sets up all the variables that are accessed in the superstructure control GUI, as well the window properties.
setupGUI()	GLFWwindow* window, render* renderAP	void	Public / ui.cpp	This function initializes AntTweakBar, giving it the current window size, and then populates the pointers for the GUI windows, then sets up the individual GUI windows with their respective functions, finally setting global options that apply to every window.