KF6017 Submission Form

Student code: **16007006**

Have you been given an extension for this assignment? **No**

Which advanced element have you completed? **Flexible component system**

What part of the engine does your dynamic model illustrate?
The object manager checking all collisions and the reaction of any intersecting objects

If you have used modelling notation other than UML, please provide a reference: **N/A**

Essay word count (excluding titles, references, footnotes and labels): **887**
Aforementioned essay begins on the following page

Additional Notes

In the case the following was not clear to the player:

- WASD controls move the player (UFO)
- Q/E fire the photons
- Rocks are indestructible and the player will die if they collide with one
- Cows are worth 100 points
- Score will increase by 1 per 60frames to reward longevity, as with an endless runner

Shell Engine Improvements: Advantages & Limitations

Advantages

From the additions I have made to the shell engine, a game programmer can now easily add different objects, and customise their behaviour by adding different components. The flexible components system enables game objects to have as many additional components as necessary, on top of the core components required by all game objects.

Building on this, the components follow a generic structure, simplifying the creation of new components and standardising their usage. Any component in the game will be a subclass inheriting from the abstract superclass “Component” and must override the pure virtual “Update” function. An exception to this is if the programmer wishes to create another abstract subclass, with its own subclasses; this is demonstrated by the family of collision components, where-in the root “CollisionComponent” remains abstract, providing only the foundations for its subclasses. Pointers to these components are stored in a list, and each time the game updates, the object manager will order all game objects to iterate through the contents of their respective component lists, calling each component’s update method.

Adding a new component to an object is as simple as calling the AddComponent() function, passing a pointer to the new component as the parameter. This can be done in a rigid manner, where object A will always be created with components X, Y & Z, but it is also possible to have this change at runtime, for example, the player may only have movement until level 5, at which point they’re handed a component which carries the ability to shoot a weapon, and at level 10 the game may introduce power-ups, and so on. In the context of a development team, this enables gameplay programmers to worry about creating new components to enhance gameplay, without needing too much (if any) discussion with the level designers – only people who work on components of the same family need to be aware of these changes, and merge conflicts are minimised. In order to further reduce dependencies, forward declarations have been used in header files wherever possible, only including the files that are directly required.

Creating a new type of GO capable of being pushed out by the object manager’s factory is also simplified. The programmer need only create a new member function to the object manager class, which takes the necessary parameters and crafts a new game object with the relevant components and attributes. In the example game, this is demonstrate with 4 functions: CreateUFO(), CreateBullet(), CreateRock() and CreateBullet(). Since each game object carries a pointer to the object manager which created it, the game object itself is able to create objects for itself, back-calling via this pointer. An example of this is the UFO creating its own bullets.

The object manager is not limited to object creation and updating, though. Using the list of game objects, it checks for inactive objects on each game update loop, and deletes them. At the end of the game, the object manager will clean up and delete all objects, regardless of their state. Game-wide collision checks are also instigated by the object manager, and then processed by the attached collision component(s).

Limitations

With generic component structure, it becomes difficult to differentiate which objects have which components, and what tasks they’re actually capable of performing. This issue is amplified if the components are changed at runtime. Solutions to this include creating a component dictionary, so the engine can browse what components a given object currently has, and access them individually. This would require exploratory programming, so the source code itself would not throw compilation errors

for including possibilities that are not necessarily reachable/possible. In the popular game engine Unity, it is possible to call GetComponent() on any game object, and operate on that component under the assumption it exists (Unity, 2018). If, at runtime, the component is not found, a null pointer is thrown and the programmer can deal with this however they wish, usually through a try-catch block.

Where components needed specific access, the game engine currently considers them "core" components, and a separate pointer is kept by the GameObject to ensure the compiler agrees they can run their own functions – the only core components are RenderComponent and CollisionComponent, which need to run LoadImg() and ProcessCollision(), respectively. Ideally, the engine would need no such guarantees, and the components could all be contained within the list.

Currently, the addition of a physics component to any object merely facilitates the calculation, rather than expanding behaviour. To improve this, the engine could offer the physics component as separate modules/functions, to be called upon and adapted to better simulate more complex movement and collisions to act as a foundation for "default" physics behaviour. Whilst it would be best to include this for any larger engine to be used in the industry, such ability is, however, achievable within the current engine, with a gameplay programmer adding their own custom component.

For a game to the scale of the example game created to demonstrate the engine, creating new functions in object manager is sufficient. In a larger project, however, it may become convoluted to have hundreds of functions inside of the ObjectManager class. It may be better practice to use these creation functions for categories, rather than individual types; replacing CreateCow() with CreateEnemy(), and then contextualising through the use of custom components at runtime.

References

Audio Micro (2018)
Photon Gun Sot
Available at: https://www.audiomicro.com/photon-gun-shot-science-fiction-sci-fi-photon-gun-shot-free-sound-effects-40803
(Accessed: 29th December, 2018)

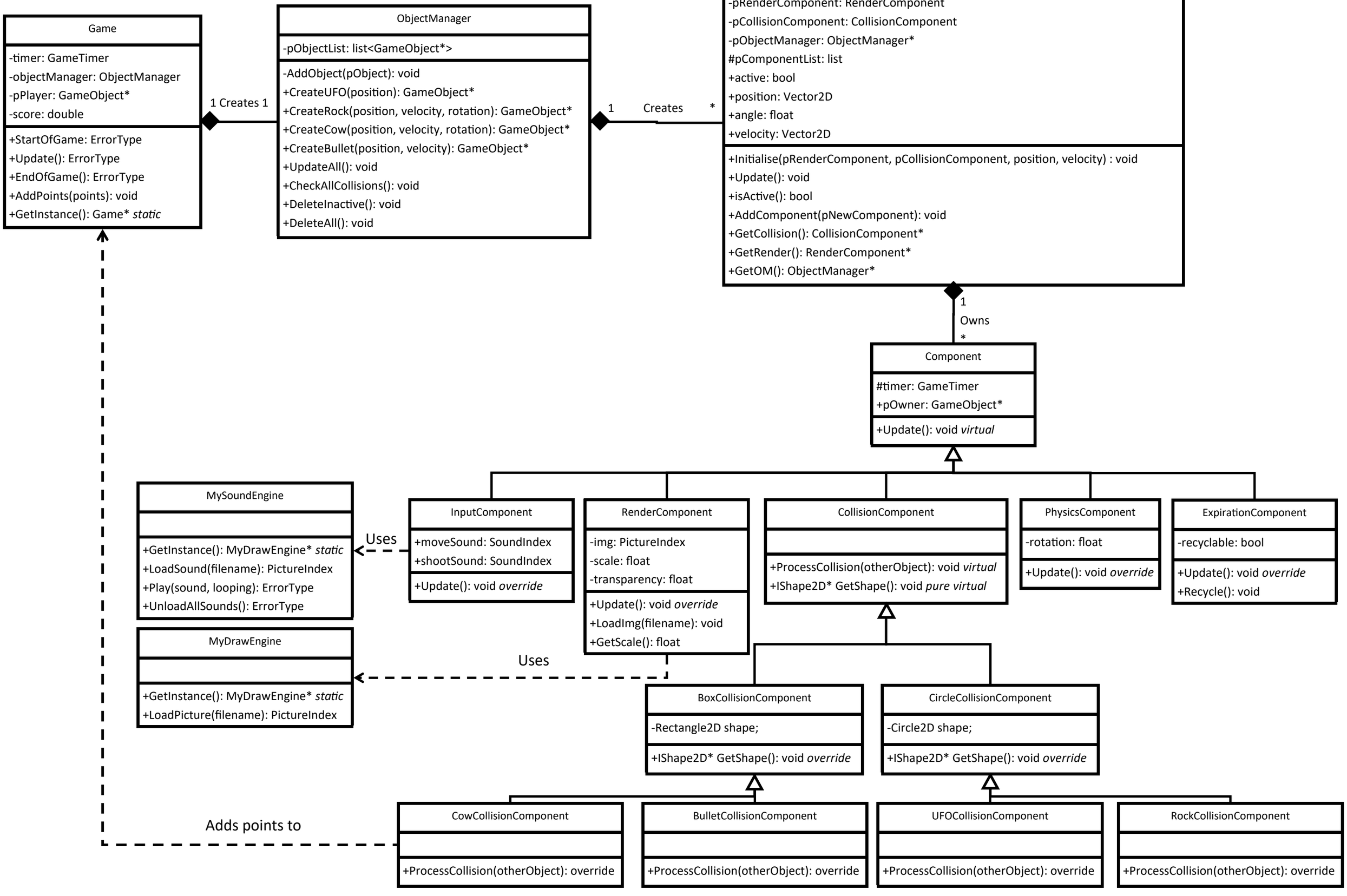
Bodzio855 (2018)
UFO Pixel Art Flying Saucer
Available at: https://pixabay.com/en/ufo-pixel-art-flying-saucer-aliens-3628773/
(Accessed: 10th December, 2018)

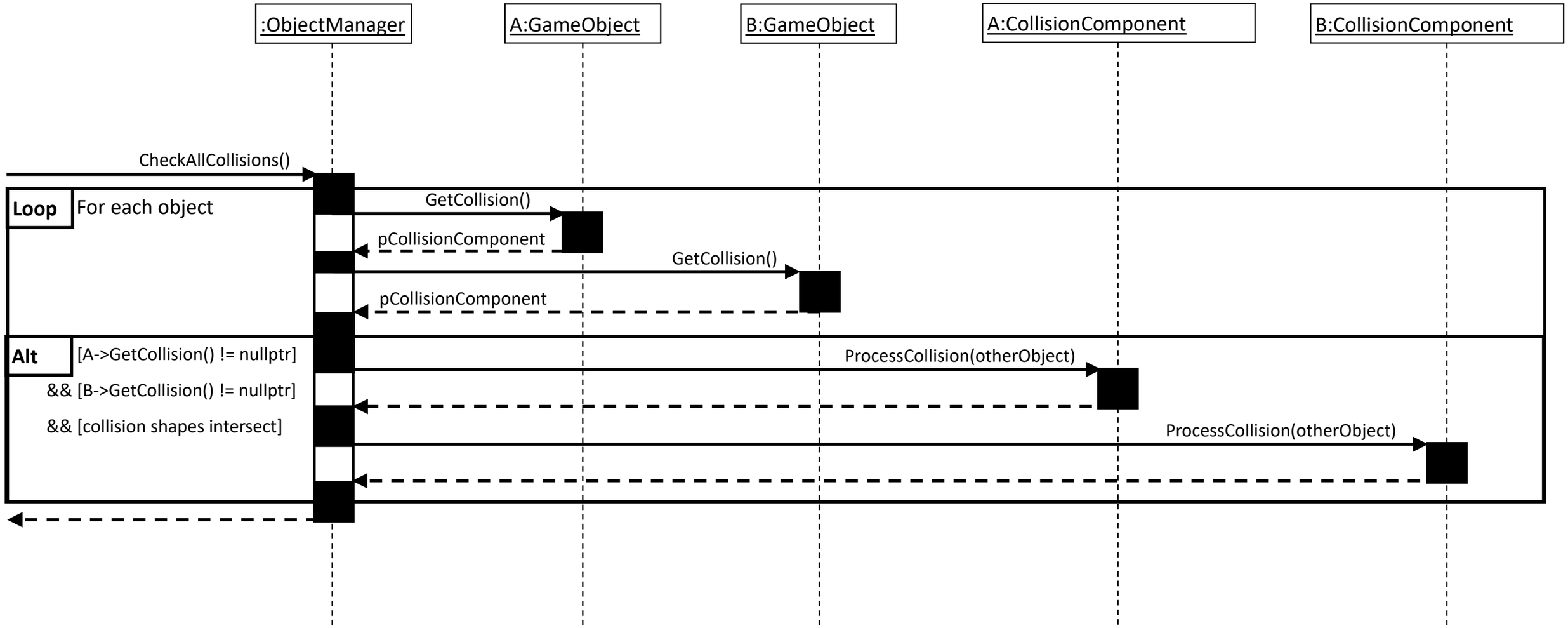
Daneeklu (2018)
LP Style Farm Animals
Available at: https://opengameart.org/content/lpc-style-farm-animals
(Accessed: 10th December, 2018)

Unity (2018)
Scripting API: GameObject.GetComponent
Available at: https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html
(Accessed: 3rd January, 2019)

Word count: 887 *(Excluding References)*

NOTE: To save space, classes provided in the base engine have only been included where relevant, and only the methods/attributes used or added by my assignment.





NOTE: Since CollisionComponent is an abstract class, the ProcessCollision function may result in further extension of this diagram, depending on the decided behaviour of the game object. This diagram is intended to show the core engine functionality which will occur in every game object, regardless of the game being programmed.

Though not communicable in this diagram, B is always the object directly after A in the ObjectManager's list.