# Explanation of the Game Engine Architecture

## Entity Component System (Component Registry)

The game engine uses an entity component system architecture (ECS). Game objects are not singular objects, but rather collections of components, some independent, some interconnected.

Components are managed by the Component Registry (ComponentRegistry class). All components are stored in a hash-table of hash-tables (see diagram 1), to which they can be registered, unregistered, or retrieved from. The Component Registry uses class inheritance and C++ templates to be fully dynamic – new components can be created simply by extending the Component superclass. There are two main virtual methods that can be overridden: Start and Update. Start is called at the beginning of the next frame after the component is registered, Update is called once every frame (after the component is registered)[[1]](#footnote-1). If the component never starts, or never updates, the virtual methods Startable and Updatable can be overridden to save on performance.

Components belonging to the same game object can be identified by the object’s UUID (universally unique identifier). The class GameObject acts as a wrapper to this UUID to make it more intuitive when accessing the Component Registry (the engine does not have a concept of a “true” game object).

There are many advantages of entity component system architecture over other game engine architectures, namely versatility, high cohesion and low coupling, use of composition over inheritance, capability of wide extension, and high performance.

* Versatility: As mentioned before, the Component Registry is fully independent on the exact types of components it can work with while providing full functionality, meaning the engine can be very easily adopted to any game (not just the one created).
* High cohesion and low coupling: Each component is self-contained and only connected to other components necessary for its functionality. Every new component naturally works with the Component Registry – no additional code needs to be added to it.
* Composition over inheritance: Rather than inheriting behaviour from parent classes, game objects are simply composed of only the specific components they require.
* Open to extension: Any new functionality can be added through components without affecting any core parts of the engine.
* High performance: The Component Registry is optimized to store components in such way to maximize cache hits – all transform components are stored together, all sprite renderer components are stored together, etc (see diagram 1). This is important, since components must be updated every frame.

The main disadvantage of entity component architecture is its complexity. Working with the Component Registry may feel unintuitive, however this is xxxxx by the helper GameObject class.

## Components

There are several engine/built-in components, such as transform, collider, or sprite renderer. These are not unique to any particular game and can be freely reused in any game that would be created with the engine. Other components relate specifically to objects used in the game.

A special component is the collider component (Component\_Collider class). Besides start and update, the collider component also provides functionality for collision management. Collisions are also handled in the Component Registry, more specifically by the CollisionManager class. Any collider components are automatically added to (and removed from) the collision manager when registered (and unregistered). Collisions between colliders are checked each frame alongside update. Component\_Collider is an abstract class and must be extended to provide a collider shape. The engine provides a circle and an (angled) rectangle colliders. These can be either used as they are, or they can be further extended into specific colliders (such as the Component\_PlayerCollider), which can implement logic for when collision occurs (through methods OnCollisionEnter, OnCollisionStay and OnCollisionLeave).

Because the Shell renders images in order they are drawn on the screen, while the Component Registry updates components in a random order, there is a separate ImageRenderer class, which manages rendering images in order of their layers. The sprite renderer and the animator components only register images to be drawn (in Update) and the ImageRenderer then draws them in the correct order at the end of the frame.

## Scenes

The game engine allows gameplay to be separated into different scenes. Each game scene has its own component registry (meaning scenes can have different sets of components). New scenes can be created by extending the abstract Scene class and overriding the Load and Unload methods. Scenes can be changed with the singleton SceneManager class.

## Game Object Factory

The game uses a game object factory to compose game objects (GameObjectFactory class). The class contains a jump table of object creation functions, which create the demanded game object by combining components and setting their values. This section of the code is inevitably messy, it would greatly benefit from a visual editor (similar to Unity or Unreal Engine). There is also no way to automate this part – each game object needs to be coded separately. A tool could be created, which would automatically write the code for the developer, if the engine was to be developed further.

## Game

The game is mostly comprised of components that control the behaviour of game objects.

1. This was inspired by the Unity Game Engine (Unity Technologies). [↑](#footnote-ref-1)