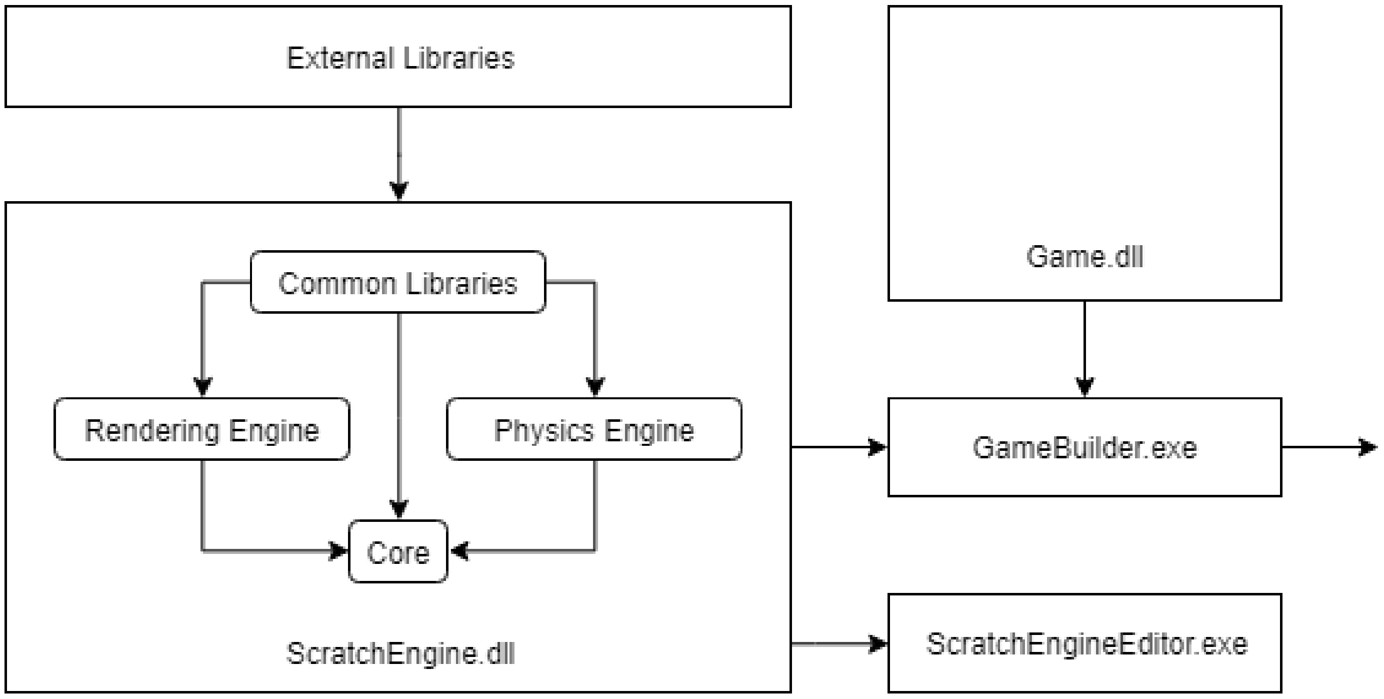
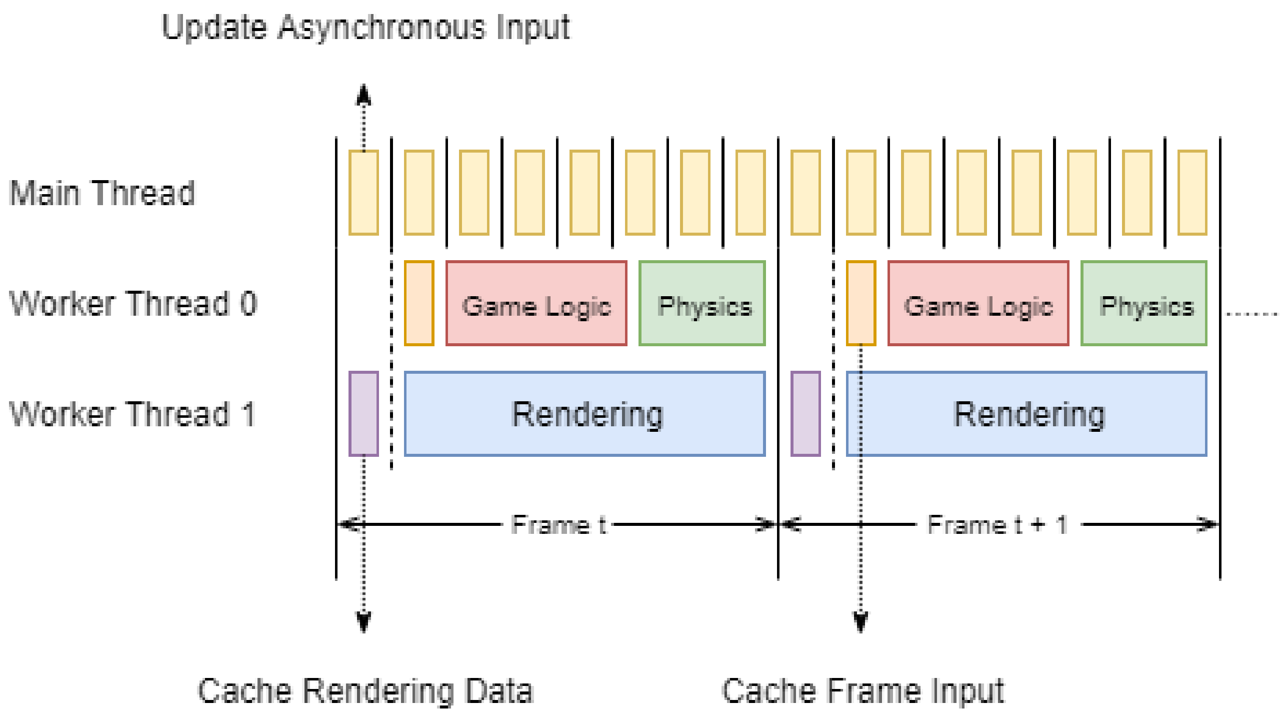
Scratch Engine

1. **Entire Architecture**



The engine itself is a .dll project, which allows both the editor program and actual games to use the same code base. Also, this design allows the whole system to be upgraded easily by only upgrading required .dll files.

2. **Systems**
   a) Multithreaded Pipeline



The engine runs with a pipelined model, in which rendering and updating are running at the same time in different thread. Currently, the engine has three threads including the main thread. Two dedicated worker threads will be created when the engine starts up and the main thread will never synchronize with those worker threads unless the engine is being shut down.

- Main Thread

The main thread is mainly used to handle Windows messages, which provides user input data for the rest of the engine. Since there is not much workload on the main thread, the input can be gathered with a really high frequency, which can minimize the latency issue.

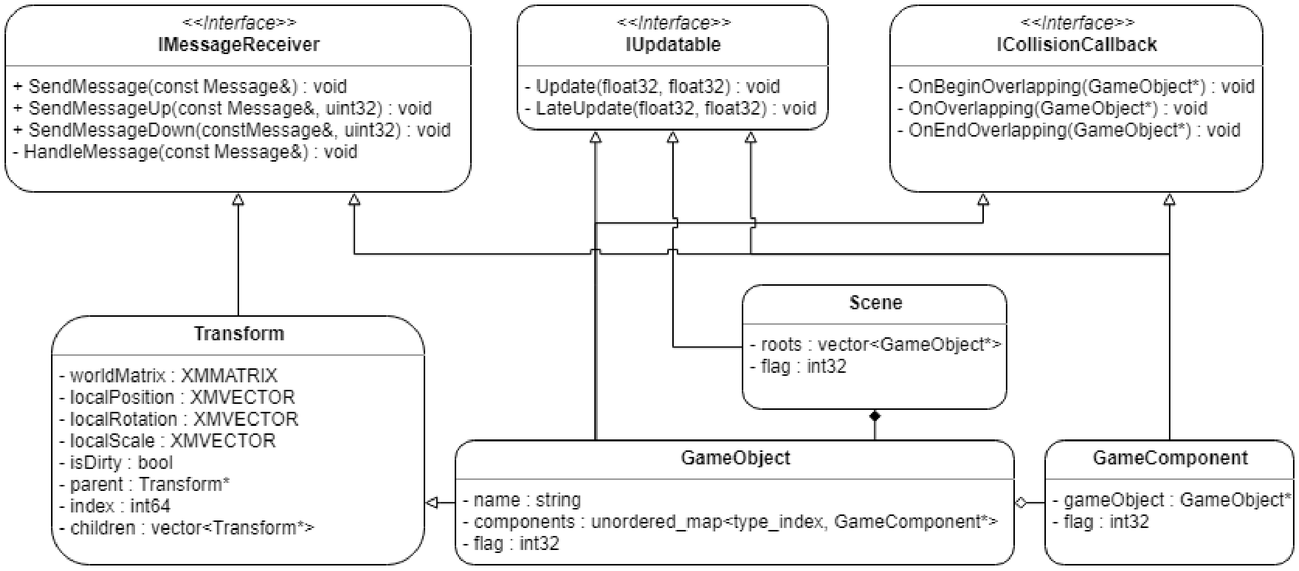
- Worker Thread 0

The first worker thread is dedicated on updating, mainly handling game logic including physics calculation. At the beginning of each frame, this thread will first wait until the rendering thread finishes copying all required data for rendering. All physics calculations are delayed after the regular game logic update.

- Worker Thread 1

The second thread is dedicated on rendering. At the beginning of each frame, the thread will first copy all rendering data into specialized structs to get rid to ensure that the updating thread will never update the data currently used by this thread. In this case, at each frame, the updating thread is doing all the logic update for that frame while this thread is rendering the last frame.

b) Entity Component System (Core)



<<Interface>>
IMessageReceiver
+ SendMessage(const Message&) : void
+ SendMessageUp(const Message&, uint32) : void
+ SendMessageDown(constMessage&, uint32) : void
- HandleMessage(const Message&) : void
<<Interface>>
IUpdatable
- Update(float32, float32) : void
- LateUpdate(float32, float32) : void
<<Interface>>
ICollisionCallback
- OnBeginOverlapping(GameObject*) : void
- OnOverlapping(GameObject*) : void
- OnEndOverlapping(GameObject*) : void
Scene
- roots : vector<GameObject*>
- flag : int32
Transform
- worldMatrix : XMMATRIX
- localPosition : XMVECTOR
- localRotation : XMVECTOR
- localScale : XMVECTOR
- isDirty : bool
- parent : Transform*
- index : int64
- children : vector<Transform*>
GameObject
- name : string
- components : unordered_map<type_index, GameComponent*>
- flag : int32
GameComponent
- gameObject : GameObject*
- flag : int32

The ECS system works in a similar way to the Unity one except the Transform is no longer a component. In each update, the engine will invoke an update call on the current scene, which will call update on every root game object. Whenever an update call is issued to a game object, the game object will recursively update its own children and components. In this way, all game components in the scene will finally be updated recursively. Similarly, the messaging system works in the same way and has specific function calls or parameters which allows users to control the depth of recursion. However, any callbacks issued through the ICollisionCallback will only affect the target game object and its components.