Shatter Space

Technical Documentation

Overview

The codebase of this game is segregated into Engine code and Game code. The engine code follows an ECS architecture. It holds the math library and other algorithms. The engine code can directly be used to create more games without any change. It is totally isolated from the game code. This Engine is inspired heavily from the Unity Game Engine.

Installation and Execution

Shatter Space was written in C++ using Microsoft Visual Studio 2022. It has been tested on C++ 14. No special step is required to install and run the game. Simply open it in Visual Studio and press Play (Start Debugging).

Code Organization

The Engine codebase has been organized in several folders. The following table provides a brief on each folder.

Folder Name	**Content Brief**
Algorithms	Holds the code for AABB and BVH implementations for efficient 3D collision detection.
Components	Holds the Entity class, Component class, and any class which derives from Component.
Core	Holds the Object class, Logger class, and utility functions.
Math	Holds the Vector, Triangle, Mesh, and Matrix classes. Includes most of the mathematical operations which are part of the Engine.
Pools	Holds the Entity pool class and its base class.
Systems	Holds the main Engine class along with all the singleton classes which work together to run the Game. It contains the Scene class as well.

The game code folder contains all Components created for the game. These components can be thought of as similar to the MonoBehavior scripts generally created in Unity projects.

System Architecture

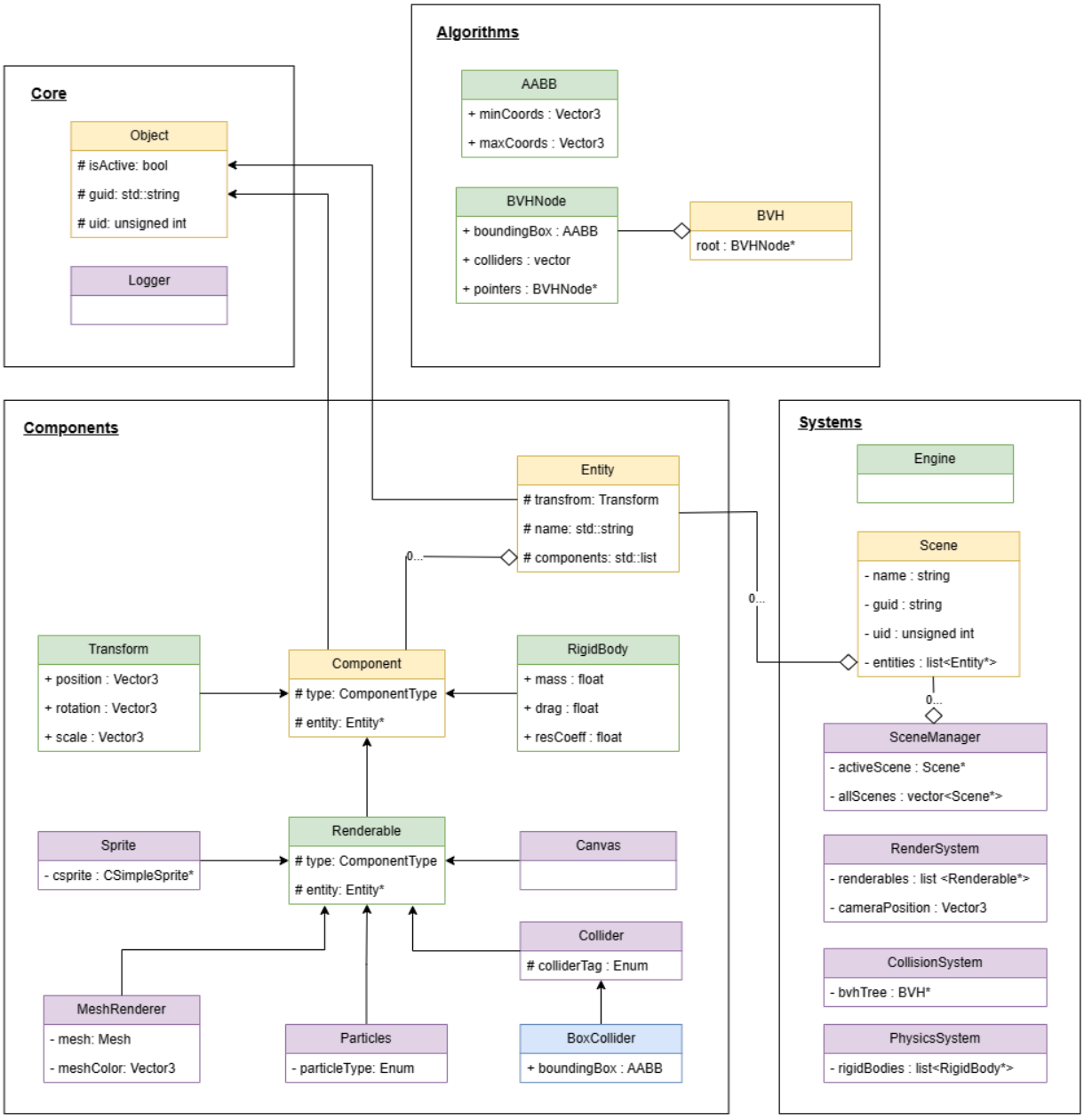
The Engine codebase uses the widely popular ECS (Entity Component System) architecture.

Salient Features

- *SceneManager* class holds all the Scenes.
- *Scene* class holds all the Entities.
- *Entity* class holds all the Components. An entity is similar to a GameObject in Unity.
    - Each component has a different purpose. The Transform component is always present in an entity. Other components can be present or not.
    - Entities can be created only via *SceneManager*, which uses *EntityPool* to get free entities.
    - Entities and Components are tightly packed in memory allowing cache coherency and preventing memory fragmentation.
- All *Components* get updated in a game loop.
- A *Renderable* is a type of Component which can be rendered on the screen.
- Different systems manage different types of components or renderables.
    - *SceneManager* updates the Scene, its Entities, and all the Components of these Entities.
    - *RenderSystem* renders all the Renderables.
    - *CollisionSystem* maintains the BVH (Bounding Volume Hierarchy) tree which is used to check for collisions.
    - *PhysicsSystem* updates all the RigidBodies. This involves kinematic and dynamic updates.

Class Diagram

Note: This diagram does not include all the classes or all the member variables and functions. It includes only the most important classes and their member variables.



Object Pooling

Object pooling is used for the entities in this game engine. Entities containing the same combination of components are stored in the same object pool. This concept is known as **Archetypes**. For example, any entity containing components A, B, C will have the same archetype and as a result would be part of the same entity pool. A new entity containing A, B, C, and D components would have a different archetype and would be part of a different entity pool. As a result, multiple object pools get created when a scene loads up. These object pools are tracked by SceneManager by creating a hash of each archetype.

Upon the creation of a new object pool, a predefined number of entities and their components automatically get allocated in the memory. This ensures that entities are tightly packed with their components in the memory, resulting in less cache misses in a game loop, which results in faster game loop updates.
Even though Scene holds the Entities, it does not have the ownership of these entities. When we remove an entity from a scene, it gets returned to its entity pool. Ultimately, the entity pool has ownership of the entities and components created by it.

Collision Detection

This game engine has only Box to Box collision detection. Boxes are stored using AABB (Axis-Aligned Bounding Box) in the BoxCollider class. Brute force collision detection has quadratic time complexity. To improve this, collision detection has been implemented using BVH (Bounding Volume Hierarchy) tree in 3D. This improves collision detection complexity to O(nlogn).
In a game, most of the entities frequently move on the screen. The BVH tree would have to be updated on each transform update of an entity that has a BoxCollider. Rebuilding the tree on each entity transform update is quite inefficient. Hence, we update only the AABBs of the nodes of the BVH tree instead of actually moving around the colliders present in these nodes. This takes logarithmic time but makes the BVH tree unbalanced over time. Hence, after enough transform updates, the BVH tree gets rebuilt.

Physics System

Any entity which has a RigidBody component will be subjected to physics updates by the PhysicsSystem. In kinematic updates, acceleration, velocity, and position of the entity get updated. In dynamic updates, any applied force gets converted to instantaneous acceleration, friction / air drag gets applied, and conservation of momentum gets applied if there's any collision.
On collision, the change in direction of velocity vector of an entity is dependent on the collision normal. The BVH tree class was modified to get the collision normal. Having box colliders stored as AABBs makes this task much simpler as the collision normals would always be parallel to the X, Y, or Z axes.

Math Systems

Some classes were written for the mathematics functionalities required in this engine. These were namely *Vector3*, *Triangle*, *Mesh*, and *Matrix4x4*. Each Triangle holds three Vector3 values which represent the edges of the triangle. A Mesh consists of a list of Triangles. The Matrix4x4 class is mainly used for the 3D to 2D projection.

Tests

Unit tests are run for different classes when the game gets executed in debug mode.