Behemoth Engine

Welcome to Behemoth Engine! This engine uses an Entity Component System (ECS) archiecture to support both 2D and 3D rendering. The ECS architecture was inpsired by a blog by the creator of the EnTT library (https://skypjack.github.io/ (https://skypjack.github.io/)). This engine also supports multithreading, physics such as collision detection, collision response, raycasting and much more. It currently is only supported for the window operating system. As mentioned earlier the Behemoth Engine is capable of loading and rendering 3D models. Currently the .obj file type is the only supported model file type. Like most ECS libraries Behemoth engine seperates data and function into components and systems. Components will only consist of data while systems will be stateless, meaning they will hold no member variables and only have member functions. Currently, only the rendering calculations are supported for multithreading however the existing architecture should allow for much more multithreading implementation in the future.

Build System

- Ensure you have premake5 installed
- From the cmd window in the main project directory run **premake5 vs2019**

Behemoth engine also contains an elementary build system for multiple games at once. To build a new game, without overwriting other data enter the following

- **premake5 vs2019 --new --prj=TestProject1**
- Replace the "TestProject1" with whatever name you with to name your new project To swap between builds simply remove the --new from the previous statement. For example
- **premake5 vs2019 --prj=TestProject1** will load an existing project named TestProject1
- Project specific files will be loaded in the **Source/Games directory**
- For ease of use, running **premake5 vs2019** will build ***Next Game*** which is the 2024 Ubisoft Next submission
- The build system will auto-generate a starter scene to help users become familiar with the engine.

Entity Component System (ECS)

The entity component system for this engine uses a registry to manage all entities and components in the scene. The only want to create and destory entities and components is via the registry. The entire ECS system is inside the **ECS** namespace, which is used throughout the engine.

How to generate a registry

```
#include "ECS/Registry.h"
ECS::Registry registry;
```

Entity Handles

Entities are commonly accessed via a `ECS::EntityHandle` which is a container for the entity ID. This struct wraps around the `ECS::Entity` class allowing for the user to copy, duplicate and store IDs. The `ECS::Entity` ID is a 32 bit integer that is split into a 16 bit version and 16 bit identifer. When an entity or component is destroyed they are recycled for later use. The version is used to ensure that if an identifier is recycled, and it matches a deleted entity, that the ECS system does not return a match for the deleted entity. This is important for later when generating groups of entities. The ECS library handles all recycling for the user, so the user does not have to worry about versions and identifiers. Entities can be optionally created with a name. This is only for debugging purposes for the user and will not affect the ECS system. Finally, entities are marked as invalid if their ID or version is set to a null value. For the entire ID the macro `NULL_ENTITY` is used indicate that an entity is not valid. This is the value `0xFFFFFFFF` or a max value for an unsigned 32 bit integer. Some functions will return a NULL_ENTITY to indicate that something has failed or not be found.

It is importantt to note that if a entity ID goes out of scope the entity is not deleted.The only way to delete an entity is via the registry's DestroyEntity function, or by deleting the registry.

Entity Creation & Destruction

```
ECS::EntityHandle handle = registry.CreateEntity("Entity Name");
registry.DestroyEntity(handle);
```

Adding Components

Components are another large part of the ECS system. They are initalized via a lazy system, where the sparse set container is created when the first component is created. Components are created via the registry and require an Entity or EntityHandle. Components are accessed via the **Sparse Set** datas structure which makes for very fast look-up and for fast cache access.

```
ECS::EntityHandle handle = registry.CreateEntity("Entity");
registry.AddComponent<Behemoth::TransformComponent>(handle);
```

Multiple types of components already exist within the engine. These components range from render, or physics components to more generic types such as `Behemoth::TransformComponent` and `Behemoth::ScalingComponent`. Components that come with Behemoth engine are within the `Behemoth` namespace. When the add component function is used it calls the constructor of the respective component. The constructor parameters are then forwarded.

```
ECS::EntityHandle entity = registry.CreateEntity("Camera");
registry.AddComponent<Behemoth::CameraComponent>(entity, isMain);
registry.AddComponent<Behemoth::VelocityComponent>(entity);
registry.AddComponent<Behemoth::TransformComponent>(entity);
registry.AddComponent<Behemoth::RotationComponent>(entity);
registry.AddComponent<Behemoth::MoveComponent>(entity, BMath::Vector3(0.0f, 0.0f, 0.0f));
```

Adding Duplicate Components

If a component is added to an entity that already possess that component, the old component is deleted and the new comonent takes it's place

Removing Components

Removing components is also very simple and requires the registry and entityHandle/ID

```
registry.RemoveComponent<PlayerFellComponent>(entity);
```

Pointer Stability

One of the most beneficial features of Behemoth's ECS system is that it supports pointer stability. This is done via paginated components. The sparse set class contains a vector of smart pointers, which point to an array of components. The page sizes for componets is defaulted to 512 but can be easily changed. When the entity identifers excced the page limit a new page is created. This avoids copying all the previous component information to a larger area of memory. Avoiding this copying also ensure that pointers and reference continue to point to the correct memory location. Unique pointers, point to each individual page and they are stored in a vector. If the vector needs to be resized then only the page pointers are moved, not the components that they point too.

```
using page_ptr = std::unique_ptr<T[]>;
inline void AddPage()
{
            pages.push_back(std::make_unique<T[]>(pageSize));
}
std::vector<page_ptr> pages;
std::size_t pageSize;
```

Pointer stability also allows us to return pointers to components on creation. However, if a component fails to be added to an entity it will return a `nullptr`.

```
Behemoth::MoveComponent* moveComponent = registry.AddComponent<Behemoth::MoveComponent>(entityHandle, BMath::Vector3(0,0,0));
```

If an entity is deleted this pointer will no longer point to the correct entity.

Recycling Components

Paginated components are recycled just like entities. The dense or packed identifer of the sparse set is used to indicate the next entity or compoent to be recycled. Essentially, the 16 bit identifer of an entity is altered to indicate the location of the next entity to be recycled. This almost creates a queue like structure but without the additional overhead of pushing, popping or allocating new memory. The only thing each sparse set requires is one `16 bit` identifer to point to the first entity to be recycled. When this entity is recycled the identifier, pointing to it's next entity is removed and stored back into next.

```
void RemoveComponent(Entity entity)
{
        entity_identifier identifier = entity.GetIdentifier();
        if (!Contains(entity))
        {
                return;
        }

        dense[sparse[identifier]].SetName("Deleted");

        // Use dense identifier to signal the next recycled entity to be reused
        if (available > 0)
        {
                dense[sparse[identifier]].SetIdentifier(next);
        }

        // Next is used to track the position of the next recycled entity to be used
        next = sparse[identifier];

        // Could use the dense identifier for this but then we would have to use the entity identifier, through the sparse to get de
        // this way we can skip one of those steps
        Entity::SetVersion(sparse[identifier], NULL_VERSION);
        available++;
}
```

Creating Custom Components

Users are free to create and add their own components but **all components are required to inheret from the ECS::Component class**

```
struct VelocityComponent : public ECS::Component
        {
                VelocityComponent() : velocity (BMath::Vector3{}) {}
                VelocityComponent(BMath::Vector3 vel) : velocity (vel) {}

                BMath::Vector3 velocity;
        };

        struct RotationComponent : public ECS::Component
        {
                RotationComponent()     :quat (BMath::Quaternion::Identity()), isAdditive(false) {}
                RotationComponent(BMath::Quaternion q, bool additive = false) : quat (q), isAdditive(additive) {}

                BMath::Quaternion quat;
                bool isAdditive;
        };
```

Get Components

It is possible to return all components of a given type. It is also possible to return groups of components. This is done via the Get() function. This is a templated function which accepts any number of component parameters. It will then return a ***vector of tuples***. It is then possible to iterate over each vector element. Get will only return entites that possess at least all of the component types passed in as template parameters. Get will always return valid pointers and therefore there is no need to check if pointers are valid.

```
for (const auto& [entity, velocityComp, transformComp] : registry.Get<VelocityComponent, TransformComponent>())
{
    ...
}
```

`Get()` first returns the entity that own these components

Systems

Systems are the last major component of the ECS architecture. It is a Singleton class that manages all systems in the game. Systems are stateless meanin they contain no data or member variables and only contain one or more member functions. Systems are required to have at least a `Run()` function that has a `const float` and `ECS::Registry&` parameters.

```
void Run(const float deltaTime, ECS::Registry& registry);
```

You will be unable to add systems to the system manager if it does not have this exact function Systems are designed to access one or more components to perform operations or logic on their data.

System Manager

The system manager is in charge of running all systems `Run()` function. For the built-in systems they are added automatically in the world initalization. Users are free to create and add their own systems. For a system to run two steps must be done.

- First: it must have the Run function listed above
- Second: it must be added to the system manager. As seen below

```
Behemoth::SystemManager::GetInstance().AddSystem<ScalingSystem>();
Behemoth::SystemManager::GetInstance().AddSystem<RotationSystem>();
Behemoth::SystemManager::GetInstance().AddSystem<MeshInitSystem>();
```

This is only required to be done once per game. If a system is added multiple times, it will still only call the `Run()` function once.

Creating and Rendering

There are some fundamental components that almost all entities posses. This includes components such as `TransformComponents` `Mesh Components` etc. It is important to note that if an entity requires a position then it must have a render component. This would be true for objects that contain meshes, colliders, lights etc.

Mesh Initalization

Any object that requires rendering must have a mesh initalization component added to it. This component will only run once before it is removed and is charge of generating the mesh data and other required components.

Object Bounds

Two bounding volumes will be generated and added to any mesh, unless specifically instructed not too via it's constructor. These bounding volumes are:

- `BoundingVolumeComponent`
- `BVHColliderComponent`

The bounding volume component is used for culling meshes entirely outside of the cameras view frustum. It uses a sphere for it's check, and will automatically generate this sphere to fit the object. The BVHColliderComponent is used for the physics collision system. This system uses a Bounding Volume Hierarchy to quickly prune meshes that are far from colliding with an object. This uses and AABB and will also auto-generate to fit the mesh. The user is not required to manually implement or add either of these components. They are automatically added when the `MeshInitComponent` is added to an entity.

Parent-Child Components

Behemoth Engine supports adding child entities to parents. The child will then have its transform automatically updated when changes are made to the parent transform. This includes scaling, rotation and movement.Adding a parent and child requires two components, and requires that the parent is provided with the childs entityID so that it can notify the child when a change in it's transform has occurred.

```
ECS::EntityHandle parentEntity = registry.CreateEntity("Parent");
Behemoth::ParentComponent* parentComponent = registry.AddComponent<Behemoth::ParentComponent>(parentEntity);

ECS::EntityHandle childEntity = registry.CreateEntity("Child");
Behemoth::ChildComponent* childComponent = registry.AddComponent<Behemoth::ChildComponent>(childEntity);

if (parentComponent)
{
        parentComponent->childHandles.push_back(childEntity);
}
```

Factory Design Pattern

Behemoth Engine utilizes a factory design pattern to easily and quickly create common game objects. By adding the registry to a factory, we let it add all the necessary components and return the `EntityHandle` which will can be used to access the components at a later date.

```
ECS::EntityHandle GameObjectFactory::CreateGameObject(
        ECS::Registry& registry,
        std::string modelFilePath,
        std::string texturePath,
        std::string entityName,
        BMath::Vector2 uvScale)
{
        ECS::EntityHandle entity = registry.CreateEntity(entityName);
        registry.AddComponent<MeshComponent>(entity, modelFilePath, texturePath, uvScale);
        registry.AddComponent<MeshInitalizeComponent>(entity);
        registry.AddComponent<TransformComponent>(entity);
        registry.AddComponent<VelocityComponent>(entity);
        registry.AddComponent<RotationComponent>(entity);

        return entity;
}
```

Common factories include the `GameObject` factory which generates a generic object with a mesh, and others such as `CameraFactory` and `LightFactory`

A factory also exists for adding a child to a parent. This is the recommended way of adding a child/parent component because if first checks if a parent already has a parent component. If it does, it will push the ID otherwise it will add a new component

Physics

Colliders

Behemoth engine supports a wide number of physics colliders and functionality. This includes:

- AABBColliders
- Sphere Colliders
- OBB Colliders

These are stored and managed automatically via the collider components.

Bounding Volume Hierarchy

Behemoth engine utalizes a bounding volume hierarchy to quickly eliminate colliders from the collision check. This is considered the **Broad** part of the collision detection process.

Narrow Collision Detection

Once the bounding volume hierarchy detects a possible collision it generates a broad collision pair component that is to be checked by the `NarrowCollisionSystem`. This is a much more through check. If a collision is found to occur it generates the contact data and adds it to a collision data component. This will later be resolved by the collision resolution system

Raycasting

Behemoth engine also supports raycast collision detection. Two functions can be used to determine either the closest collision, or all collisions.

```
bool RayCast(ECS::Registry& registry, const Ray& ray, std::vector<ContactData>& data, const std::vector<ECS::EntityHandle>& entitiesT
bool RayCast(ECS::Registry& registry, const Ray& ray, ContactData& data, const std::vector<ECS::EntityHandle>& entitiesToIgnore, BMa
```

Collision Masks

Collision masks are implemented to ensure only desired collisions are processed.

Math Library

Behemoth engine supports a wide variety of Vectors, Matricies and Quaternions. Various checks have also been implemented to ensure accuraccy with the math library