Hym Game Engine - Overview

DaOnlyOwner

March 16, 2022

Contents

1 Goal Overview

- This is my first engine so I will use plenty of libraries to support the development
- Engine for small scale scenes (no open world)
- Uses modern graphics APIs (Vulkan / DirectX 12 through Diligent Engine)
- Uses innovative rendering approaches (Realtime GI through raytracing, GTAO etc.)
- Nothing is static, everything can move without performance penalties.
- CPU memory allocation: Uses the normal allocator. In the future this will be optimized.
- Deferred rendering is used.

2 Class Overview Public Interfaces

2.1 ResourceManager

This class represents the resource manager. It manages GPU resources, such as vertex buffers or textures. Because the engine targets small scale scenes, we can afford loading all the scene resources into memory, static geometry, as well as dynamic resources that can despawn. This allows for the optimization that every mesh can be stored into one big vertex buffer and index buffer. When a mesh is spawned into the world, the spawned entity holds the location (*index*) of the start of its first vertex and index in the buffer. This can only be done because resources are not unloaded. If unloading would occur, the buffer shrinks (otherwise what's the point) and the locations (the indices) need to be updated for every spawned mesh. Same goes for textures. This allows for bindless resources which will increase performance. The second point is that a global buffer is needed in order to compute ray tracing data (i.e. getting the normal of the hitpoint, or sampling the texture of the hit object).
The interface will support loading of meshes and textures. When loading a i.e. fbx file you can specify an alias name (i.e. *scene1*) for that scene contained in the file. To retrieve an object named for example *obj1* from the scene file you ask the manager to return the model with identifier *scene1* and *obj1*. If you want to retrieve the whole scene, the manager will return a vector of meshes from input *scene1*. The ResourceManager defines a Model as a Mesh together with a collection of textures. You will get back the model when asking for it. Unloading: As stated, unloading shouldn't be necessary, the workflow in the editor might require deleting a model entirely from the resources. In this case unloading is supported. Note that this shouldn't be done at runtime because it incurs a substantial performance cost.

ResourceManager Interface

- *LoadSceneFile(String filename, String alias);*
- *Model GetModel(String sceneAlias, String objectName);*
- *vector<Model >GetSceneModels(String sceneAlias);*
- *UnloadModel(String sceneAlias, String objectName, ArrayRef<Scene >scenesToUpdate);*
- *UnloadScene(String sceneAlias, ArrayRef <Scene >) scenesToUpdate)*

2.2 Concept

A concept is similar to Unity's Prefab system. It is a collection of components that work together. Each component contains data, like position, render state

etc. This data is then used in the systems of the engine, like the renderer. The interface supports adding and removing components from the concept. A concept can only hold one component of each type.

Concept Interface

- *AddComponent <Comp >(Comp componentToAdd);*
- *Comp* GetComponent <Comp >();*
- *DelComponent<Comp >();*
- *GetID();* Returns the id of the component.

2.3 Scene

This class represents the scene. You can add a concept to the scene, if it has a model component and a transform component, the model will be displayed at the position defined in the transform component. When you add a concept, the scene returns an ID. You can use the ID as a way to handle state of the concept, such as updating components or removing the concept from the scene again. The scene can be serialized and deserialized.

Scene Interface

- *Scene(ResourceManager* resManager);*
- *Id AddConcept(const Concept* c);* This function copies the contents of c into the scene and returns the new Id.
- *Concept* GetConcept(Id conceptId);*
- *EnableSun();*
- *DisableSun();*
- *Sun* GetSun();*
- *DelConcept(Id conceptID);*
- *Serialize(String path);*
- *Deserialize(String path);*

2.4 Sun

The sun represents the directional light. You can set the time of day and the skysphere, color of the sunlight and direction of the sunbeams will adapt to the changes. Additionally you can set the direction and color yourself.

2.5 Camera

This class represents a physical camera in the world. You can retrieve the view and projection matrix from here, as well as set the eye position and the lookAt point. FOV etc. can be specified here.

2.6 Transform Component

This struct contains the transform of the entity. You can retrieve the model matrix and the normal matrix from this component.

Scene Interface

- Field: *vec3 rotation*;
- Field: *vec3 position;*
- Field: *vec3 scale;*
- *mat4 GetNormal_ModelMatrix();*

2.7 Model Component

This component holds the mesh and a collection of textures. The mesh holds a vertex and index offset into the global buffers as well as the size of its indices. (For indexed drawing). The textures are just locations in the global texture buffer.

2.8 PointLight Component

WIP

2.9 SpotLight Component

WIP

2.10 AreaLight Component

WIP

2.11 Renderer

WIP