Grok3d

An Ecs game engine written in c++ with excessive documentation

**Design**

1. ECS game engine
2. Made up of entities, components, component mangers, and systems

**World**

* System that ties it all together, use this to get the EntityManager, ComponentManagers, and systems
* Use it to tell an entity to be removed
* Essentially a reference holder for all the managers
* These managers include things not needing to be part of a component. Eg input module, time. OS interface

**Entities**

* Essentially a typdef’d ID wrapped in a **EntityHandle**
* A handle simple contains the entity (a number) and a pointer to the world
  + Has a addcomponent method who calls world->addComponent(entity, compoenent). World knows which **Component Manager** to send it to by the type of component (templates), define a in world getComponentManager<ComponentType>();
  + Has destroy, calls world->destroy(entity)
* *It is possible to have a quick check if an entity has certain components by*
  + *Storing a bitvector of components where each bit is one type of component*
  + *Giving bits their component value using templates like so ClassManager here https://github.com/bibekdahal/ecs-engine/blob/c68d13713ce3f5c7446114d0b10506c6158a20d8/Component.h*
  + *The same thing could be done for global event types! Now I don’t have to name them all I can just do ComponentManager::GetComponentID<ComponentType>() which simply increments for each new one or EventManager::GetEventID<EventType>()*
* Should AddComponent return something for errors or should we utilize exceptions?

**Components**

* Data only pertaining to certain functionality eg health, position, speed, is physical
* all members should be some implementation of **Cvars** so they can be updated as a string from debug console
* Component is handled with a **ComponentHandle** who contains the Entity who owns it, a pointer to the component itself, a pointer to it’s manager, this is what is returned by ComponentManager::Lookup(Entity e) method

**Component Managers**

* Generic class which contains vector of components of that type
* Data is stored in a hashmap of **entities** to **ComponentInstances**
* **ComponentInstances** are indexes of an internal vector of the templated **Component** type (eg HealthComponent)
* Adding a component to an entity is simple, call the manager’s add function with entity ID argument and *(component reference argument || arguments needed to make component)*. This then adds the entityID to componentInstance map, where componentInstance’s value is the current size of the internal storage array of components (because that’s the next free index)
* Removing is a bit more complex…remove it from the hashmap as well as take the last element of the internal storage vector, put it where the deleted component was, and update that individual in the above entityID to componentInstance map.
* UpdateAll method that takes a lambda and updates all
  + This can be used to update all, certain components, etc

**Systems**

* *Systems should have some way of knowing what it should pay attention to*, be it when all the necessary components are added or manually adding as well as references to any entities they may need to write to/access in a system. EG poison cloud has ref to player so when player is within radius of poison cloud(can call getLocationCompoenent on player and calc distance) it then calls getHelathCompoenent and decs the health
  + Can be done when world adds compoenents are added/removed from entity
  + Every system has registerEntityIfNeeded on addComponent from world function, inline function in system (CRTP or functions not in class) checks if bitmask of entity now matches
  + When a component is added to entity it goes through all the systems that are now “on” because the mask dictated it
  + It calls registerEntity on that system for the entity the componenet is being added to
* System accesses whatever it needs to, checks if all the necessary components are there for what it needs to do, and acts on the data based on its internal rules.
* The game loop iterates over every system (some in a specific order) to update game state once per tick
* All systems which do an update take in a change in time **dt** and use that, everything is based on **time not frame**
* Example: Player movement system. Entity with location, player control, and velocity is updated locationx = dt\*xvelocity\*xdir and same for y where xdir is -1 1 or 0.
* System **update** and **render** functions are decoupled. You advance the simulation in increments of a set value (the tickrate) and do the physics steps that number of times, saving values less than the tickrate in time on an accumulator, then render, and repeat checking time difference and adding to accumulator. See <https://gafferongames.com/post/fix_your_timestep/>

**Callback/Message system**

* Callback system for systems entities or components to respond to
  + Component creation
  + Update
  + Render
  + Component delete
  + Etc.
  + OnCollisionEnter
  + OnCollisionExit
  + OnHealthEmpty
* Messages for more logic based functionality, message is a string and parameter list, each GameLogic component can handle however they want, applies to this object and its children.
* Entities can be organized in a tree, send messages to parents who can respond by sending messages down.

**Console System**

* Keeps a log of everything “printed” for debug purposes
* As mentioned has ability to edit CVARS

**Network**

Use a circular list in the network and input system, when server updates, roll back to timestamp and reapply the inputs on the list. Bigger list, less rubber banding

**Initialization**

* Initialize by creating components and entities to put them in and then letting it go by the rules
* Can be done in a function in c++
* Can be a XML/JSON/SOMETHINGELSE file that is parsed and the objects listed are created in their given state (saves can also be implemented by creating these)

**Engine wide Events**

https://github.com/bibekdahal/ecs-engine/blob/c68d13713ce3f5c7446114d0b10506c6158a20d8/Component.h

The same thing could be done for global event types! Now I don’t have to name them all I can just do ComponentManager::GetComponentID<ComponentType>() which simply increments for each new one or EventManager::GetEventID<EventType>()

**GameLogic events**

What i'm thinking is something akin to a subscriber/listener pattern where there are some global events (or even events pertaining to different kinds of entities).  EG there is a button with an Activatable component (or script component or whatever), this component has an event, you create a door that subscribes to that event, when button is pressed it may do quite a few things.  The alternative is to just do this all statically and not dynamically within the Activatable's Activated function.  I think being able to add to a list of functions is way better, why not? Activatable can have a "trigger" that IT subscribes to (eg a collision with the player, or the player pressing e, calls a function which checks if you're close enough and looking at it)

So how would i structure an event system like that? I'll use c#y syntax for brevity but I guess i would just have a playerEntity reference in my GameLogic componenet constructor and i'd say

 playerHandle.GetComponent<RigidBody>().OnCollision += [thisEntityHandle] () { this.DoorBeginOpen(); }  //no idea right now how i'd handle the door opening over time but I guess i'd do something like

**Multithreading**

* Certain world level systems run on their own thread or as an actor (developer console, audio, rendering the state)
* Systems that act on the data can break their entities into chunks and send them off to be done in a thread pool. *It is important to block on these if another system relies on their results*

**Memory Allocation**

* use new at first, maybe make a stack allocator data structure that loads and unloads, make smart pointers just for deallocating in opposite order easily

**Object Heirarchy**

Abstract GameObject

Name

Destroy()

Clone()

Abstract Component : GameObject

entity

Update(dt)

GetComponent<TYPE>() gets compent of TYPE for the same entity

ReceiveMessage(string, …)

Entity : GameObject

**Component descriptions**

* Transform
  + XYZ position,
  + Rotation
  + scale
* RigidBody, contains all properties for physics calculations, extends PhysicalBody
  + Mass
  + Drag
  + HasGravity
  + Position/rotation that updates transform AFTER physics is calculated
* Collider, component for collision detection, extended by BoxCollider etc
  + attachedRigidBody
  + dimensions
  + Has isTrigger for scripting
* Camera
  + FOV
  + Background
  + Culling mask
  + Clipping planes
  + Viewport rect
* AudioSource
* GameLogic
  + Has hooks to be implemented as a script
  + Extending classes should keep track of state of what pertains to them and update those other things accordingly (eg in update if (doorIsOpening) door.move())
  + ~~Coroutines?~~
  + ~~GameLogic component implements this. Call StartCoroutine and create coroutine class of a function~~
  + <https://blogs.msdn.microsoft.com/oldnewthing/20080812-00/?p=21273>
* Velocity? Should extend GameLogic
* Acceleration? Should extend GameLogic
* Health? Should extend GameLogic
* etc

TIDBITS FROM UNITY

* Unity creates initialization of public members that are other GameObjects for you, so when you mess with other GameObjects, it’s because they’re assigned to public fields you have available by the setup before your script’s init/update functions are called.
* You can tell what you’re looking at with the camera with Raycast
  + https://unity3d.com/learn/tutorials/projects/lets-try-assignments/lets-try-shooting-raycasts-article?playlist=41639

Stretches

Scripting interface to a higher level language like LUA

My own memory management that ensures more cache hits