Aleksi Anderson

Entity Component System Implementation

Tietorakenteet ja Algoritmit

10.11.2022

Table of Contents

[1 Introduction 1](#_Toc119433544)

[2 Implementation 2](#_Toc119433545)

[2.1 Entity 2](#_Toc119433546)

[2.1.1 Entity Manager 3](#_Toc119433547)

[2.2 Component 4](#_Toc119433548)

[2.2.1 Component Array 4](#_Toc119433549)

[2.2.2 Component Manager 6](#_Toc119433550)

[2.3 System 8](#_Toc119433551)

[2.3.1 System Manager 8](#_Toc119433552)

[2.4 Integration 10](#_Toc119433553)

[3 Application 11](#_Toc119433554)

[4 Conclusion 13](#_Toc119433555)

[Sources 14](#_Toc119433556)

# Introduction

ECS or Entity Component System is a method of programming game objects by having an **entity** represent the object, **components** which contain all the data of the object, and **systems** which operate on the data in the components.

This is generally a better approach to designing game objects than the traditional inheritance approach. First, it is much more flexible than inheritance, since you assign specific components to objects and don’t have to include everything that comes before in the inheritance hierarchy. It is also better for performance, since it only has to load relevant data into the cache, instead of the entire object and all its members.

# Implementation

As stated before, Entity Component System is a method of organizing data into entities, components, and systems. It is not to be thought of as a system of entities and components.

## Entity

An entity is simply an id, which can be represented by an unsigned integer. An alias can make them easier to implement. This id is then used to index arrays containing information from every entity created to get the data for the desired entity. Since every entity must have a unique id, using a 16-bit unsigned integer has a maximum of 65535 entities. Of course, this limit can be increased by using a bigger integer, a 64-bit unsigned int would have a maximum of 18446744073709551615 entities.

//Alias Entity as a 16-bit unsigned int

using Entity = uint16\_t;

//The entity is just a number id, same as uint16\_t e;

Entity e;

Since entities are only an id, we need some way of knowing what components it has. For that we can use a signature, which is a number that tells us what components the entity has. A simple way to do this is with a bitset where each bit represents a component. For example, if Transform was 0, Collider was 1 and Sprite was 2, an entity with the Transform and Sprite components would have a signature 0b101. This means we also need to assign an id to each component.

//The maximum unique components possible,

//needed for array sizing

const uint8\_t MAX\_COMPONENTS = 1000;

//Entity's signature, a bitset with one bit representing //every possible component

using Signature = std::bitset<MAX\_COMPONENTS>;

### Entity Manager

To give each entity a unique id, we need a system to distribute and manage them. The Entity Manager is a class that does just that. In this case we use a simple stack to store the ids, when a new entity is created the top id is taken, and when it is destroyed the now free to use id is pushed to the top. The signatures are stored in an array of bitsets where the signature of a component is accessible by indexing the array with the entity. entitySignatures[entity] would be the signature of entity. There should only be one instance of EntityManager.

class EntityManager

{

public:

std::stack<Entity> availableEntityIds;

Signature entitySignatures[MAX\_ENTITIES];

uint16\_t entityCount = 0;

EntityManager()

{

//Initialize availeableEntityIds to include every possible id

for (size\_t i = 0; i < MAX\_ENTITIES; i++)

{

availableEntityIds.push(i);

}

}

//Return a unique entity id

Entity newEntity()

{

if (entityCount >= MAX\_ENTITIES)

{

std::cout << "Too Many Entities!" << std::endl;

return NAN;

}

entityCount++;

Entity e = availableEntityIds.top();

availableEntityIds.pop();

return e;

}

//Set the entity id as available

void deleteEntity(Entity e)

{

entitySignatures[e].reset();

entityCount--;

availableEntityIds.push(e);

}

};

## Component

Components are simply containers of data; they have no functions or any way of manipulating that data. A simple Transform component could be a struct with three values. Every component also has an id, which can just be an unsigned int. The id is the number of the bit in an entity’s signature bitmap. For example, an entity with signature 0b001001 would “contain” the components with ids 0 and 3.

struct Transform

{

Vector2 position;

Vector2 scale;

float rotation;

};

### Component Array

For systems to effectively find and iterate through components they need to act upon, we need to create a separate array for every type of component containing every instance of that component. These arrays should also be packed, meaning there should be no holes of unused data in it, so we could traverse the entire array without checking for valid data. This would be simple to accomplish if we didn’t have to delete components, since you could simply have an array with the entity being the index to the component. However, this does not work if you delete a component, since then there would be a hole in the array.

To solve this, we can use a map connecting the entity to its component in the packed array. Indexing the map by the entity would give the actual index of its component in the packed array. This also needs to go the other way, so we need to create a map from the component to its entity. So, to access an entity’s component we do: componentArray[componentToInxed[component]].

When deleting a component, we need to fill up the hole in the position it occupied. To do this we move the last component in the array to its position. Then we need to update both the component to index and index to component maps so that they no longer contain the deleted component. We also need to update the map for the previously last component, which we moved to fill the gap.

//Generic component array interface for component manager

class IComponentArray

{

public: virtual void removeComponent(Entity entity) = 0;

};

template<typename T>

class ComponentArray : public IComponentArray

{

public:

//Packed array of all components of type T

T componentArray[MAX\_ENTITIES];

//Map from an entity id to the index of its component in the componentArray

std::map<Entity, int> entityToIndex;

//Map from a components index in the componentArray to its entity's id

std::map<int, Entity> indexToEntity;

//Amount of components in existance

int size = 0;

void addComponent(Entity entity, T component)

{

//Update entity and index maps to include new entity

entityToIndex[entity] = size;

indexToEntity[size] = entity;

//Add the component to the packed array

componentArray[size] = component;

size++;

}

void removeComponent(Entity entity) override

{

if (entityToIndex.find(entity) == entityToIndex.end())

{

std::cout << "Trying to remove non-existent component" << std::endl;

return;

}

//Keep track of the deleted components index, and the entity of the last //component in the array

int deletedIndex = entityToIndex[entity];

Entity lastEntity = indexToEntity[size - 1];

//Overwrite the deleted component by moving the last component in the

//component array to its index

componentArray[deletedIndex] = componentArray[size - 1];

//Update the maps of the moved component

entityToIndex[lastEntity] = deletedIndex;

indexToEntity[deletedIndex] = lastEntity;

//Remove the deleted entity, and moved component from the maps

entityToIndex.erase(entity);

indexToEntity.erase(size - 1);

}

T& getComponent(Entity entity)

{

//Return a reference to entity's component

return componentArray[entityToIndex[entity]];

}

};

### Component Manager

Since we create one component array per component type, we need a class to contain and manage all the component arrays. The jobs of the component manager will be to assign unique ids to components, and to select and act on the component array of the desired type when adding or removing a component from an entity.

To assign unique ids we can simply have a counter increment each time a component is registered and use its value for the id, since we don’t have to delete or unregister components. This means that registerComponent() will have to be called for every component type. To make this easier for the user we can check if the component is registered every time it is added to an entity, and if it is not, we simply register it then. This means the user can add a custom component type to their game without having to register it manually.

The component manager must also work as an interface to all the component arrays. This means the component manager needs a map with all the component arrays accessible by their type name. This also means we need ComponentArray to inherit from a generic class so we can store multiple templated types in the same list.

All this greatly simplifies accessing and adding/removing components since we can simply call getComponent<type>(entity) similarly to how it is done in Unity.

See ECSCore.h for complete ComponentManager class

class ComponentManager

{

public:

uint16\_t nextId = 0;

//A map from a component type's name to its id

std::map<const char\*, uint16\_t> typeToId;

//A map from a component type's id to its name

std::map<uint16\_t, const char\*> idToType;

//A map from a component type's name to the component array of its type

std::map<const char\*, std::shared\_ptr<IComponentArray>> componentArrays;

//Register component type

template<typename T>

void registerComponent()

{

//Name of the new component's type

const char\* componentType = typeid(T).name();

//Assigns an id and makes a new component array for the registered

//component type

typeToId.insert({ componentType, nextId });

idToType.insert({ nextId, componentType });

componentArrays.insert({componentType, std::make\_shared<ComponentArray<T>>()});

nextId++;

}

//Adds a component of type T to entity

template<typename T>

void addComponent(Entity entity, T component)

{

//Call the addComponent method of the correct component array

getComponentArray<T>()->addComponent(entity, component);

}

void destroyEntityData(Entity entity, Signature signature)

{

for (size\_t i = 0; i < signature.size(); i++)

if (signature[i] == 0)

componentArrays[idToType[i]]->removeComponent(entity);

}

private:

//QOL function to get the casted ComponentArray

template<typename T>

std::shared\_ptr<ComponentArray<T>> getComponentArray()

{

const char\* componentType = typeid(T).name();

//If the component has not been registered, do it

if (componentArrays.find(componentType) == componentArrays.end())

{

registerComponent<T>();

}

//Return a Cast ComponentArray of desired type

return std::static\_pointer\_cast<ComponentArray<T>>(

componentArrays[componentType]);

}

};

## System

Systems are any functions that act upon entities with certain components. Every system needs a list of entities that it acts upon. This list should only include entities with the required components, which we can check with the signature. In this case using a set is better than a list since every entity is unique and should only appear once.

//Base class all systems inherit from

class System

{

public:

//Set of every entity containing the required components for the system

std::set<Entity> entities;

};

### System Manager

The system manager is a class for registering systems and holding their signatures, as well as updating each system’s entity sets when necessary. The signatures of each entity are stored in a map. They tell us what components a system requires, just as the signature of an entity tells us what components it has. Because the signatures are simple bitsets, to determine if an entity has all the required components for a system is as simple as a bitwise and operation.

For example, system A requires components with ids 2, 5, and 6, thus the signature of system A is 0b110010. Entity E has components with ids 1, 2, 4, and 6, thus the signature of entity E is 0b101011. A bitwise and (&) of these two signatures will return a signature with every component they both have in common, in this example it would be 0b100010. Therefore, if this signature is the same as the system’s signature, we know that the entity has every required component. In this example however, the resulting signature is not the same as the system’s signature, meaning the entity does not have every required component, which is true since it is missing component #5.

Each time an entity changes its signature, meaning a component gets added or removed, we must update the set of entities in every system so that they only contain entities with the required components, this is where the bitset signatures come in handy.

class SystemManager

{

public:

//Map of all systems accessible by their type names

std::map<const char\*, std::shared\_ptr<System>> systems;

//Map of each system's signature accessible by their type name

std::map<const char\*, Signature> systemSignatures;

template<typename T>

std::shared\_ptr<T> registerSystem()

{

//Name of the new system's type

const char\* sytemType = typeid(T).name();

//Create new system and return a pointer to it

std::shared\_ptr<T> system = std::make\_shared<T>();

systems.insert({ sytemType, system });

return system;

}

//Sets the signature (required components) for the system

template<typename T>

void setSignature(Signature signature)

{

//Name of the new system's type

const char\* sytemType = typeid(T).name();

systemSignatures.insert({sytemType, signature});

}

void destroyEntity(Entity entity)

{

//Loop through each system and remove the destoyed entity

for (auto const& system : systems)

{

system.second->entities.erase(entity);

}

}

void onEntitySignatureChanged(Entity entity, Signature entitySignature)

{

//Loop through every system

for (auto const& system : systems)

{

//Bitwise and to check if the entity contains all the required

//components for the system

if ((entitySignature & systemSignatures[system.first]) ==

systemSignatures[system.first])

{

//Add the entity to the system's set

system.second->entities.insert(entity);

}

else

{

//Remove the entity from the system's set

system.second->entities.erase(entity);

}

}

}

};

## Integration

Now that we have entities, components, and systems, along with their managers, we need some convenient way to communicate with all of them. A good way to do this is to have a class which contains an instance of each manager and coordinates all three. With this we don’t have to manually tell each manager if we want to, for example, add a component to an entity instead we just tell the ECS manager class.

See ECSCore.h for complete ECS class

class ECS

{

public:

EntityManager\* entityManager;

ComponentManager\* componentManager;

SystemManager\* systemManager;

ECS()

{

componentManager = new ComponentManager();

entityManager = new EntityManager();

systemManager = new SystemManager();

}

~ECS()

{

delete componentManager;

delete entityManager;

delete systemManager;

}

//This class also contains numerous methods to interface with every

//manager class. It was just far too long to fit here

};

# Application

Below is a small example creating a falling entity with a very rudimentary gravity system:

First, we define the two required components, and the system in a separate file (or same file if you really want to). Every function of the system should loop through every entity in its list because systems are only meant to be instanced once and they should act on every entity with the required components. Because we have the system manager keep track of what system should act on what entity, there is no need to validate the entity in the system function.

//Simple position and gravity Components

struct Position

{

float x, y;

};

struct Gravity

{

float x, y;

};

//Get the ecs instance from the main file

extern ECS ecs;

//System class, this holds every function of the GravitySystem system

class GravitySystem : public System

{

public:

//Update the entity's position

void Update()

{

//For each entity that has the required components

for (auto const& entity : entities)

{

//Get the relevant components from entity

Position& position = ecs.getComponent<Position>(entity);

Gravity& gravity = ecs.getComponent<Gravity>(entity);

//Update the entity's postion component

position.y += gravity.y;

position.x += gravity.x;

}

}

};

With ECS don’t need to have any logic code in the main loop, just like with inheritance hierarchies. Here we must first create a global instance of the ecs manager, then we register the gravity system, then we create a signature, set it to require the Position and Gravity components and attach it to the GravitySystem. Finally, we create a new entity called player and add the Position and Gravity components to it. Now when we call GravitySystem.Update(), it updates the position of every entity with the Position and Gravity components.

//Create one instance of the ecs manager

ECS ecs;

int main()

{

//Register the gravity system, it is accessible by this pointer

std::shared\_ptr<GravitySystem> gravitySystem =

ecs.registerSystem<GravitySystem>();

//Add Position and Gravity components as requirements for GravitySystem

Signature gravitySystemSignature;

gravitySystemSignature.set(ecs.getComponentId<Position>());

gravitySystemSignature.set(ecs.getComponentId<Gravity>());

ecs.setSystemSignature<GravitySystem>(gravitySystemSignature);

//Create a new entity

Entity player = ecs.newEntity();

//Add the position component and set it's starting position

ecs.addComponent(player, Position{.x = 4, .y = 10});

//Add the gravity component and set it's direction

ecs.addComponent(player, Gravity{ .x = 0, .y = -1.0f });

while (true)

{

//Gravity system updates every entity with the required components

gravitySystem->Update();

std::cout << "Y: " << ecs.getComponent<Position>(player).y << ", X: " << ecs.getComponent<Position>(player).x << std::endl;

}

}

# Conclusion

In conclusion, ECS is an alternative to inheritance hierarchies when structuring game objects. It provides greater flexibility in making different objects and utilizes the CPU cache better, which is the main performance bottleneck in object-heavy games. It works by diving data and functionality into entities, which are simply ids, components, which are plain old data, and systems which contain functions to operate on data.

Sources

https://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

https://austinmorlan.com/posts/entity\_component\_system/