ENTITY COMPONENT SYSTEM (ECS)

**source**: <https://austinmorlan.com/posts/entity_component_system/>

ECS is an architectural pattern used in game development; it consists of 3 parts:

1. **Entity**. Every object in your game is an entity. It is an abstract concept not represented by any data structure. It is just an *ID* number, there is no ‘*class Entity’.* The ID is used as an index into an array of components.
2. **Components.** All the data about your entities are in components. A component is a *dumb data* container without any executable functions. Sometimes represented as a struct.
3. **Systems.** All your game logic code is in systems. A system might hold data which is relevant to the system itself, but no data about the entities it possesses (this is a components job)

ECS follows *composition over inheritance*, which allows greater flexibility in defining entities.

Traditionally in game development, you would follow an inheritance approach to problems. A goblin would inherit from a monster, which would inherit from an actor. A shopkeeper would inherit from a human, which would inherit from an actor. The actor class would contain a render() method.

Example using OOP (inheritance):

ENTITY

STATIC DYNAMIC

ROCK TREE ENEMY PLAYER

??

??

EVIL TREE

*Should evil tree inherit from enemy, or tree?*

Example using ECS:

TREE ENEMY PLAYER

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Position** | **AI** | **Sprite** | **AABB** | **Velocity** | **Input** |
| *Float x*  *Float y* | *Bool doLeft*  *Bool doRight*  *Bool doJump* | *Image\* layers[]*  *Float times[]* | *Float w*  *Float h* | *Float x*  *Float y* | *Bool keyLeft*  *Bool keyRight*  *Bool keyJump* |

*AABB = Axis Aligned Bounding Box* (collision detection)

* In a typical implementation, each different component type will derive from an abstract component class, which provides facilities for getting a components types and containing entity at run time.
* A system operates on related groups of components, i.e. components that belong to the same entity, e.g. the character movement system might operate on a position, velocity, collider and an input. Each system will be updated once per frame in a logical order.
  + E.g. to make a character jump, first the key jump field of the input data is checked. If true, the system will look through the contacts contained in the collider data and check there is one with the ground, if so, it will set the velocities *y field* to make the character jump.
  + Some examples of systems include:
    - movementSystem (position, velocity)
    - renderSystem(position, sprite)
    - playerControlSystem(input, player)

THE ENTITY

Using Entity = std::uint32\_t; // a simple type alias

Const Entity MAX\_ENTITIES = 5000; // used to define the size off arr later on

THE COMPONENT

Struct Transform {

Vec3 position;

Quat rotation;

Vec3 scale;

}

Using ComponentType = std::uint8\_t; //a simple type alias

Const ComponentType MAX\_COMPONENTS = 32; // used to define size of arr later on

Each component type (transform, RigidBody, etc) also has a unique ID given to it.

THE SIGNATURE

Since an entity is simply an ID, we need a way to track which components an entity ‘has’, and we also need a way to track which components a system cares about. Each component type has a unique ID (starting from 0), which is used to represent a bit in the *signature*. E.g. if *Transform* has type 0, *RigidBody* has type 1, and *Gravity* has type 2, an entity that ‘has’ those 3 components would have a signature of 0b111 (bits 0, 1, and 2 are set).

Using Signature = std::bitset<MAX\_COMPONENTS>

THE ENTITY MANAGER

The *EntityManager* is in charge of distributing entity IDs and keeping record of which IDs are in use, and which are not.

Class EntityManager {

Public:

EntityManager() {

// initialize the queue with all possible entity IDs

For(Entity entity = 0; entity < MAX\_ENTITIES; ++entity) {

mAvailableEntities.push(entity);

}

}

Entity CreateEntity() {

Assert(mLivingEntityCount < MAX\_ENTITIES && “too many entities”);

// take an ID from the front of the queue

Entity id = mAvailableEntities.front();

mAvailableEntities.pop();

++mLivingEntityCount;

Return id;

}

Void DestroyEntity(Entity entity) {

Assert(entity < MAX\_ENTITIES && “entity out of range”);

// invalidate the destroyed entities signature

mSignature[entity].reset();

// put the destroyed ID as the back of the queue

mAvailableEntities.push(entity);

--mLivingEntityCount;

}

Void SetSignature(Entity entity, Signature signature) {

Assert(entity < MAX\_ENTITIES && “entity out of range”);

// put this entitys signature in to the array

mSignatures[entity] = signature;

}

Signature GetSignature(Entity entity) {

Assert(entity < MAX\_ENTITIES && “entity out of range”);

// get this entities signature from the array

Return mSignature[entity];

}

Private:

Std::queue<Entity> mAvailableEntities{}; // queue of unused entity IDs

Std::array<Signature, MAX\_ENTITIES> mSignature{}; // array of signatures

Uint32\_t mLivingEntityCount{}; // total living entities

}

THE COMPONENT ARRAY

If an entity is just an index into an array of components, then it is simple to grab the relevant component for an entity, but what happens when an entity is destroyed? That index into the array is no longer valid.

One way to solve this is by mapping from entity IDs to array indices. When accessing the array, you use the entity ID to look up the actual array index. Then, when an entity is destroyed, you take the last valid element in the array and move it into the deleted entity’s spot and update the map so that the entity ID now points to the correct position.

1. Let’s assume that MAX\_ENTITIES is set to 5. The array starts out empty.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 0 | 1 | 2 | 3 | 4 |
| Entity->index |  |  |  |  |  |
| Index->Entity |  |  |  |  |  |
| Size: **0** |  |  |  |  |  |

1. We then add a component with value A to Entity 0. Entity 0 maps to index 0, and index 0 maps to Entity 0

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 0  A | 1 | 2 | 3 | 4 |
| Entity->index | 0:0 |  |  |  |  |
| Index->Entity | 0:0 |  |  |  |  |
| Size: **1** |  |  |  |  |  |

1. We add some further values

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 0  A | 1  B | 2  C | 3  D | 4 |
| Entity->index | 0:0 | 1:1 | 2:2 | 3:3 |  |
| Index->Entity | 0:0 | 1:1 | 2:2 | 3:3 |  |
| Size: **4** |  |  |  |  |  |

1. Now, we delete the value *B* from Entity 1. To keep it packed, we move the last element D into the spot occupied by *B*, and update the maps. Entity 3 maps to index 1, and index 1 maps to Entity 3.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 0  A | 1  D | 2  C | 3 | 4 |
| Entity->index | 0:0 | 2:2 | 3:1 |  |  |
| Index->Entity | 0:0 | 1:3 | 2:2 |  |  |
| Size: **3** |  |  |  |  |  |

1. We then delete the value *D* from Entity 3, moving the last element *C* into the spot occupied by *D*. Entity 2 maps to index 1, and index 1 maps to Entity 2.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 0  A | 1  C | 2 | 3 | 4 |
| Entity->index | 0:0 | 2:1 |  |  |  |
| Index->Entity | 0:0 | 1:2 |  |  |  |
| Size: **2** |  |  |  |  |  |

1. Finally, we add value *E* to Entity 4. Entity 4 maps to index 2, and index 2 maps to Entity 4.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Array | 0  A | 1  C | 2  E | 3 | 4 |
| Entity->index | 0:0 | 2:1 | 4:2 |  |  |
| Index->Entity | 0:0 | 1:2 | 2:4 |  |  |
| Size: **3** |  |  |  |  |  |

Implementation

Class IComponentArray {

Public:

Virtual ~IComponentArray() = default;

Virtual void EntityDestroyed(Entity entity) = 0;

}

Template<typename T>

Class ComponentArray : public IComponentArray {

Public:

Void InsertData(Entity entity, T component) {

Assert(mEntityToIndexMap.find(entity) == mEntityToIndexMap.end() && “component added to same entity more than once”);

// put new entry at and update the maps

Size\_t newIndex = mSize;

mEntityToIndexMap[entity] = newIndex;

mIndexToEntityMap[newIndex] = entity;

mComponentArray[newIndex] = component;

++mSize;

}

Void RemoveData(Entity entity) {

Assert(mEntityToIndexMap.find(entity) != mEntityToIndexMap.end() && “Removing non-existent component.”);

// copy element at end into deleted elements place to maintain density

Size\_t indexOfRemovedEntity = mEntityToIndexMap[entity];

Size\_t indexOfLastElement = mSize – 1;

mComponentArray[indexOfRemovedEntity] = mComponentArray[indexOfLastElement];

// update map to point to moved spot

Entity entityOfLastElement = mIndexToEntityMap[indexOfLastElement];

mEntityToIndexMap[entityOfLastElement] = indexOfRemovedEntity;

mIndexToEntityMap[indexOfRemovedEntity] = entityOfLastElement;

mEntityToIndexMap.erase(entity);

mIndexToEntityMap.erase(indexOfLastElement);

--mSize;

}

T& GetData(Entity entity) {

Assert(mEntityToIndexMap.find(entity) != mEntityToIndexMap.end() && “retrieving non-existent component”);

Return mComponentArray[mEntityToIndexMap[entity]];

}

Void EntityDestroyed(Entity entity) {

If (mEntityToIndexMap.find(entity) != mEntityToIndexMap.end()) {

removeData(entity);

}

}

Private:

Std::array<T, MAX\_ENTITIES> mComponentArray;

Std::unordered\_map<Entity, size\_t> mEntityToIndexMap;

Std::unordered\_map<size\_t, Entity> mIndextoEntityMap;

Size\_t mSize;

}

THE COMPONENT MANAGER

The ComponentManager is responsible for talking to all the different ComponentArrays when a component needs to be added or removed.

Class ComponentManager {

Public:

Template<typename T>

Void RegisterComponent() {

Const char\* typeName = typeid(T).name();

Assert(mComponentTypes.find(typeName) == mComponentTypes.end() && “Registering component type more than once”);

// add this component type to the component type map

mComponentTypes.insert({typename, mNextComponentType});

// create a ComponentArray pointer and add it to the component arrays map

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

// increment the value so that the next component registered will be different

++mNextComponentType;

}

Template<typename T>

Component GetComponentType() {

Const char\* typeName = typeid(T).name();

Assert(mComponentTypes.find(typeName) != mComponentTypes.end() && “Component not registered before use”);

// return this components type – used for creating signatures

Return mComponentTypes[typeName];

}

Template<typename T>

Void AddComponent(Entity entity, T component) {

// add a component to the array for an entity

GetComponentArray<T>()->InsertData(entity, component);

}

Template<typename T>

Void RemoveComponent(Entity entity) {

// remove a component from the array for an entity

GetComponentArray<T>()->RemoveData(entity);

}

Template<typename T>

T& GetComponent(Entity entity) {

// get a reference to a component from the array for an entity

Return GetComponentArray<T>()->GetData(entity);

}

Void EntityDestroyed(Entity entity) {

// Notify each component array that an entity has been destroyed

// If it has a component for that entity, it will remove it

For (auto const& pair : mComponentArrays) {

Auto const& component = pair.second;

Component->EntityDestroyed(entity);

}

}

Private:

// Map from type string pointer to a component type

Std::unordered\_map<const char\*, ComponentType> mComponentTypes{};

// Map from type string pointer to a component array

Std::unordered\_map<const char\*, std::shared\_ptr<IComponentArray>> mComponentArrays{};

// the component type to be assigned to the next registered component – starting at 0

ComponentType mNextComponentType{};

// convenience function to get the statically casted pointer to the ComponentArray of type T

Template<typename T>

Std::shared\_ptr<ComponentArray<T>> GetComponentArray() {

Const char\* typeName = typeid(T).name();

Assert(mComponentTypes.find(typeName) != mComponentTypes.end() && “Component not registered before use”);

Return std::static\_pointer\_cast<ComponentArray<T>> (mComponentArrays[typeName]);

}

}

THE SYSTEM

A system is any functionality that iterates upon a list of entities with a certain signature of components.

Class System {

Public:

Std::set<Entity> mEntities;

}

Each system can inherit from this class which allows the SystemManager to keep a list of pointers to systems. A system could then do something like:

For (auto const& entity : mEntities) {

Auto& rigidBody = GetComponent<RigidBody>(entity);

Auto& transform = GetComponent<Transform>(entity);

Auto const& gravity = GetComponent<Gravity>(entity);

transform.position += rigidBody.velocity \* dt;

rigidBody.velocity += gravity.force \* dt;

}

A RigidBody, Transform and a Gravity will be pulled into the cache for this entity as well as for all of the entities near it in the component array, which are likely to be needed with the next entity in the list of entities.

THE SYSTEM MANAGER

The SystemManager is in charge of maintaining a record of registered systems and their signatures. When a system is registered, it’s added to a map with the same typeid(T).name() trick used for the components.

Each system needs to have a signature set for it so that the manager can add appropriate entities to each system’s list of entities. When an entities signature has changed (due to components being added or removed), then the systems list of entities that its tracking needs to be updated.

class SystemManager {

public:

template<typename T>

std::shared\_ptr<T> RegisterSystem() {

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

assert(mSystems.find(typeName) == mSystems.end() && "Registering system more than once.");

// Create a pointer to the system and return it so it can be used externally

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

mSystems.insert({typeName, system});

return system;

}

template<typename T>

void SetSignature(Signature signature)

{

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

assert(mSystems.find(typeName) != mSystems.end() && "System used before registered.");

// Set the signature for this system

mSignatures.insert({typeName, signature});

}

void EntityDestroyed(Entity entity)

{

// Erase a destroyed entity from all system lists

// mEntities is a set so no check needed

for (auto const& pair : mSystems)

{

auto const& system = pair.second;

system->mEntities.erase(entity);

}

}

void EntitySignatureChanged(Entity entity, Signature entitySignature)

{

// Notify each system that an entity's signature changed

for (auto const& pair : mSystems)

{

auto const& type = pair.first;

auto const& system = pair.second;

auto const& systemSignature = mSignatures[type];

// Entity signature matches system signature - insert into set

if ((entitySignature & systemSignature) == systemSignature)

{

system->mEntities.insert(entity);

}

// Entity signature does not match system signature - erase from set

else

{

system->mEntities.erase(entity);

}

}

}

private:

// Map from system type string pointer to a signature

std::unordered\_map<const char\*, Signature> mSignatures{};

// Map from system type string pointer to a system pointer

std::unordered\_map<const char\*, std::shared\_ptr<System>> mSystems{};

};

THE COORDINATOR

The coordinator is used so the 3 managers; EntityManager, ComponentManager & SystemManager, can talk to each other.

The coordinator has pointers to each manager and does some meta-managing between them.

class Coordinator {

public:

void Init()

{

// Create pointers to each manager

mComponentManager = std::make\_unique<ComponentManager>();

mEntityManager = std::make\_unique<EntityManager>();

mSystemManager = std::make\_unique<SystemManager>();

}

// Entity methods

Entity CreateEntity()

{

return mEntityManager->CreateEntity();

}

void DestroyEntity(Entity entity)

{

mEntityManager->DestroyEntity(entity);

mComponentManager->EntityDestroyed(entity);

mSystemManager->EntityDestroyed(entity);

}

// Component methods

template<typename T>

void RegisterComponent()

{

mComponentManager->RegisterComponent<T>();

}

template<typename T>

void AddComponent(Entity entity, T component)

{

mComponentManager->AddComponent<T>(entity, component);

auto signature = mEntityManager->GetSignature(entity);

signature.set(mComponentManager->GetComponentType<T>(), true);

mEntityManager->SetSignature(entity, signature);

mSystemManager->EntitySignatureChanged(entity, signature);

}

template<typename T>

void RemoveComponent(Entity entity)

{

mComponentManager->RemoveComponent<T>(entity);

auto signature = mEntityManager->GetSignature(entity);

signature.set(mComponentManager->GetComponentType<T>(), false);

mEntityManager->SetSignature(entity, signature);

mSystemManager->EntitySignatureChanged(entity, signature);

}

template<typename T>

T& GetComponent(Entity entity)

{

return mComponentManager->GetComponent<T>(entity);

}

template<typename T>

ComponentType GetComponentType()

{

return mComponentManager->GetComponentType<T>();

}

// System methods

template<typename T>

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

{

return mSystemManager->RegisterSystem<T>();

}

template<typename T>

void SetSystemSignature(Signature signature)

{

mSystemManager->SetSignature<T>(signature);

}

private:

std::unique\_ptr<ComponentManager> mComponentManager;

std::unique\_ptr<EntityManager> mEntityManager;

std::unique\_ptr<SystemManager> mSystemManager;

};

EXAMPLE DEMO

components

struct Gravity {

Vec3 force;

};

struct RigidBody {

Vec3 velocity;

Vec3 acceleration;

};

struct Transform {

Vec3 position;

Vec3 rotation;

Vec3 scale;

};

A system for rudimentary physics integration

extern Coordinator gCoordinator;

void PhysicsSystem::Update(float dt) {

for (auto const& entity : mEntities)

{

auto& rigidBody = gCoordinator.GetComponent<RigidBody>(entity);

auto& transform = gCoordinator.GetComponent<Transform>(entity);

auto const& gravity = gCoordinator.GetComponent<Gravity>(entity);

transform.position += rigidBody.velocity \* dt;

rigidBody.velocity += gravity.force \* dt;

}

}

The main loop

Coordinator gCoordinator;

int main() {

gCoordinator.Init();

gCoordinator.RegisterComponent<Gravity>();

gCoordinator.RegisterComponent<RigidBody>();

gCoordinator.RegisterComponent<Transform>();

auto physicsSystem = gCoordinator.RegisterSystem<PhysicsSystem>();

Signature signature;

signature.set(gCoordinator.GetComponentType<Gravity>());

signature.set(gCoordinator.GetComponentType<RigidBody>());

signature.set(gCoordinator.GetComponentType<Transform>());

gCoordinator.SetSystemSignature<PhysicsSystem>(signature);

std::vector<Entity> entities(MAX\_ENTITIES);

std::default\_random\_engine generator;

std::uniform\_real\_distribution<float> randPosition(-100.0f, 100.0f);

std::uniform\_real\_distribution<float> randRotation(0.0f, 3.0f);

std::uniform\_real\_distribution<float> randScale(3.0f, 5.0f);

std::uniform\_real\_distribution<float> randGravity(-10.0f, -1.0f);

float scale = randScale(generator);

for (auto& entity : entities)

{

entity = gCoordinator.CreateEntity();

gCoordinator.AddComponent(

entity,

Gravity{Vec3(0.0f, randGravity(generator), 0.0f)});

gCoordinator.AddComponent(

entity,

RigidBody{

.velocity = Vec3(0.0f, 0.0f, 0.0f),

.acceleration = Vec3(0.0f, 0.0f, 0.0f)

});

gCoordinator.AddComponent(

entity,

Transform{

.position = Vec3(randPosition(generator), randPosition(generator), randPosition(generator)),

.rotation = Vec3(randRotation(generator), randRotation(generator), randRotation(generator)),

.scale = Vec3(scale, scale, scale)

});

}

float dt = 0.0f;

while (!quit)

{

auto startTime = std::chrono::high\_resolution\_clock::now();

physicsSystem->Update(dt);

auto stopTime = std::chrono::high\_resolution\_clock::now();

dt = std::chrono::duration<float, std::chrono::seconds::period>(stopTime - startTime).count();

}

}