Fort-Bishop → Entity Component System

2.1 Entity Component System Description (/src/ECS)

What is ECS → An ECS (Entity Component System) is a system of organizing your game into entities. Each entity has components and other properties, allowing you to create almost anything with it in an **organized** manner depending on the range of **components** you have. Everything from **UI** to the **Player** can be an Entity.

Components → **Components** are very important to an ECS. The **components** are what make the ECS useful. A **TileComponent** allows the tilemap to be made of **entities**, therefore have collision. A **SpriteComponent** allows **entities** to be visible on screen.

Important Components → Three very important components are the **TransformComponent**, **SpriteComponent**, and the **HitboxComponent**. The **TransformComponent** holds multiple position-related variables like **position**, **velocity**, **scale**, and some other things. The **SpriteComponent** allows the entity to be seen on screen by rendering a sprite (even an animated one). The **HitboxComponent**, in combination with an **std**::**map** allows collisions between entities.

Manager → The **manager** simplifies doing tasks like **updating** and **rendering entities**. Instead of having to update every entity in the **Game**::**update** method, you can call one singular method to update all entities, making the code much more readable. The same thing goes for all other tasks.

2.2 (Simplified) Entity Component System Layout

Reference Next Page For Content

Manager
↳ Entities
  ↳ Components
    ↳ Constructor(s) and Destructor
    ↳ init method
    ↳ update method
    ↳ draw method
    ↳ Other methods
    ↳ Variables
    ↳ Other Code
  ↳ Methods
  ↳ Variables
  ↳ Other Code
↳ Methods
↳ Variables
↳ Other Code

2.3 Code

ECS.h (cut down, actual code in “/src/ECS/ECS.h”)

```
class Component {
    public:
        Entity* parent;
        int status;

        Component() = default;
        virtual ~Component() = default;

        virtual void init() {}

        virtual void draw() {}

        virtual void update() {}
};

// Base "Entity" class
class Entity {
    private:
        Manager &manager;

        // Bitsets
        ComponentBitset compBitset;
        GroupBitset groupBitset;

        // Array and pointers
        ComponentArray compArr;
        std::vector<std::unique_ptr<Component>> components;

        bool active = true;

    public:
        // Constructors
```

```
        Entity(Manager& mManger) : manager(mManger) {}

        // Adds component to component array
        template<typename T, typename... TArgs> T& addComponent(TArgs&&... args) {
            // Create Component
            T* c(new T(std::forward<TArgs>(args)...));

            // Parents component
            c->parent = this;

            // Adds component to components array
            std::unique_ptr<Component> uPtr {c};
            components.emplace_back(std::move(uPtr));

            compArr[getComponentTypeID<T>()] = c;
            compBitset[getComponentTypeID<T>()] = true;

            // Inits the component and returns pointer to component
            c->init();
            return *c;
        }

        // Checks to see if a certain type of component is in component array
        template<typename T> bool hasComponent() const {
            // Returns the bitset (bool value)
            return compBitset[getComponentTypeID<T>()];
        }

        // Returns a certain type of component from component array
        template<typename T> T& getComponent() const {
            // Returns pointer to component
            return *static_cast<T*>(compArr[getComponentTypeID<T>()]);
        }

        // Goes through all the components (for this entity) calls their update method
        void update() {
            for(auto &c : components) c->update();
        }

        // Goes through all the components (for this entity) calls their draw method
        void draw() {
            for(auto &c : components) c->draw();
        }

        // Returns active boolean
        inline bool isActive() {return active;}

        // Checks if entity is in specified group
        bool hasGroup(Group mGroup) {
            return groupBitset[mGroup];
        }

        // Adds entity to group
        void addGroup(Group mGroup);

        // Removes entity from group
        void delGroup(Group mGroup) {
            groupBitset[mGroup] = false;
        }

        // Sets active to false
        void destroy() {active = false;}
};

// Manager class
```

```
class Manager {
    private:
        // Vector of entites (not pointers)
        std::vector<std::unique_ptr<Entity>> entities;

        // Array of group vectors of Entity pointers
        std::array<std::vector<Entity*>, MAX_GROUPS> groupedEntities;

    public:
        // Goes through all the entities and calls their update function
        void update() {
            for(auto &e : entities) e->update();
        }

        // Goes through all the entities and calls their draw function
        void draw() {
            for(auto &e : entities) e->draw();
        }

        // Refreshes ECS
        void refresh() {
            // Refreshes groups
            for(auto i(0u); i < MAX_GROUPS; i++) {
                auto &v(groupedEntities[i]);
                v.erase(
                    std::remove_if(
                        std::begin(v),
                        std::end(v),
                        [i](Entity* mEntity) {
                            return !mEntity->isActive() || !mEntity->hasGroup(i);
                        }
                    ),
                    std::end(v)
                );
            }

            // Refreshes entities
            entities.erase(
                std::remove_if(
                    std::begin(entities),
                    std::end(entities),
                    [](const std::unique_ptr<Entity> &mEntity) {
                        return !mEntity->isActive();
                    }
                ),
                std::end(entities)
            );
        }

        // Adds entity to group (array[vector])
        void addToGroup(Entity *mEntity, Group mGroup) {
            groupedEntities[mGroup].emplace_back(mEntity);
        }

        // Returns a list of pointers to all entities in a certain group
        std::vector<Entity*> &getEntitiesFromGroup(Group mGroup) {
            return groupedEntities[mGroup];
        }

        // Creates, adds, and returns entity (use this when creating an entity so it is compatible with manager
system)
        Entity &addEntity() {
            // Creates and adds a new Entity
            Entity *e = new Entity(*this);
            std::unique_ptr<Entity> uPtr {e};
```

```
            entities.emplace_back(std::move(uPtr));
            return *e;
        }
};
```