CMPUT 350 - ECS
Entity Component Systems

Michael Buro, Jake Tuero

Department of Computing Science, University of Alberta
Edmonton, Canada
tuero@ualberta.ca

October 28, 2024

Change Log

Motivation

Suppose we want to create a video game which has the following:

- Different classes of characters types (monsters, humans, etc.)
- Each class has different races (goblin and spiders are monsters)
- All characters need to be drawn on screen and has various behaviours

What techniques have we learned to structure our code logic to reduce code complexity and promote code reuse?

Motivation

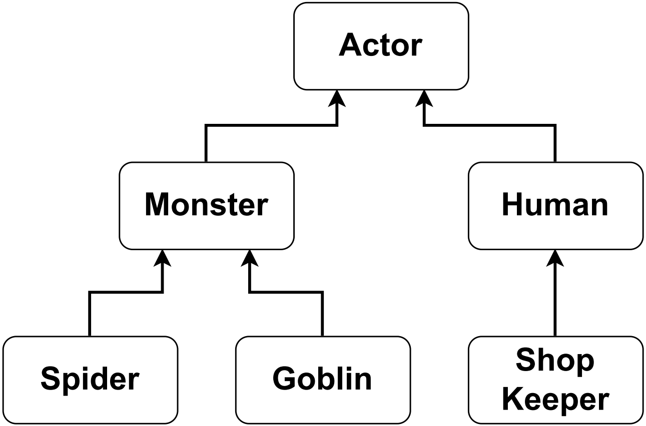
Suppose we want to create a video game which has the following:

- Different classes of characters types (monsters, humans, etc.)
- Each class has different races (goblin and spiders are monsters)
- All characters need to be drawn on screen and has various behaviours

What techniques have we learned to structure our code logic to reduce code complexity and promote code reuse?

Inheritance!

Motivation



Motivation

```cpp
struct Actor {
    virtual void Render() const = 0;
    virtual void Act() = 0;
};

// Monsters
struct Monster : public Actor { };
struct Spider : public Monster {
    void Render() const override;
    void Act() override;
};
struct Goblin : public Monster {
    void Render() const override;
    void Act() override;
};

// Humans
struct Human : public Actor { };
struct ShopKeeper : public Human {
    void Render() const override;
    void Act() override;
};
```

Motivation

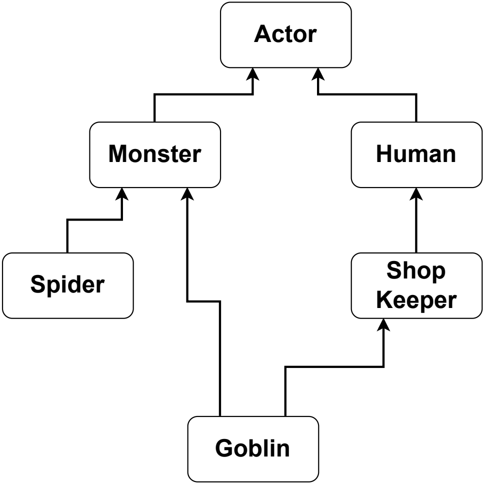
We can call the correct class behaviour through virtual functions:

```
void Render(const std::vector<Actor*> &actors) {
    for (const auto & actor : actors) {
        actor->Render();
    }
}

void Act(std::vector<Actor*> &actors) {
    for (const auto & actor : actors) {
        actor->Act();
    }
}
```

Inheritance Problems

Suppose Goblins can also be Shop-Keepers... Do they also now become Humans?

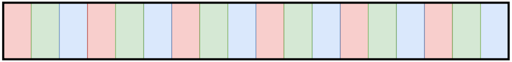


Cache Efficiency

Suppose Actors have the following data:

```
struct Actor {
    std::pair<int, int> pos;    // pos.x = red, pos.y = green
    double velocity;            // blue
};
void update_velocity(std::vector<Actor*> &actors) {
    for (const auto & actor : actors) {
        update(actor.velocity);
    }
}
```

If we many Actors (Goblins, Spiders, Shop Keepers, etc.) allocated contiguously, a function which operates on the actors will fetch them into cache lines:



Since only one data member is being operated on in this function, we are wasting 2/3 of our cache with unrelated data!

Motivation

Benefits:

- Code reuse
- Works well for tree-based dependencies

Downsides:

- How do we deal with multiple inheritance
  - Are Goblins now Humans?
- Need to use virtual function tables + pointers → object data not in cache (SLOW!)

Entity Component Systems (ECS)

Entity Component Systems are comprised of three parts:

- **Entity**: An identifier for each entity
- **Components**: Characterizes an entity as possessing a particular aspect, and holds the data to represent that.
- **Systems**: A process which acts on all entities with the desired components

ECS Example

	Entity 1	Entity 2	Entity 3
Component 1	[Data]		
Component 2	[Data]	[Data]	[Data]
Component 3		[Data]	[Data]

[1]

[1] https://en.wikipedia.org/wiki/Entity_component_system

ECS - Entity

Entities are very simple: An unqiue ID which we can use to index various groups of data

```
// size_t is often used to index contiguous data structures
using Entity = std::size_t;

// Our engine can at most handle 5000 entities
const std::size_t MAX_ENTITIES = 5000;
```

ECS - Components

Components represent a collection of data related to a particular aspect

```cpp
template <typename T>
struct Heading2D {       // 2D vector heading
    T x;
    T y;
};

template <typename T>
struct Position {        // 2D grid position
    T x;
    T y;
};

template <typename T>
struct Health {          // All units have health
    T health;
};

template <typename T>
struct Radius {          // All units have a radius
    T radius;
};
```

ECS - Components

To refer to components later, we will assign each a unique ID to it

```
// size_t is often used to index contiguous data structures
using ComponentType = std::size_t;

// Our engine can at most handle 32 components
const std::size_t MAX_COMPONENTS = 32;
```

ECS - Components

Systems will want to refer to a collection of components which define some aspect:

- Our collision system will want to update all entities which have a `Position`, `Radius`, and `Heading2D` component (non-inclusive)

As such, we need a way track which components belong to which entities (which is just an integer). We can do so by using a collection of N bits to represent which of the N components are required (either for a system or for which an entity has):

```
// Signature represents MAX_COMPONENTS bits, like our Set from Assignment 1
using Signature = std::bitset<MAX_COMPONENTS>;
```

ECS - Entity Manager

The `EntityManager` is responsible for distributing Entity IDs and tracking what is the signature of each Entity (what components does it have):

```
class EntityManager {
public:
    EntityManager();

    // Take ID from front of the queue
    //  if we have not handed out all our entities
    auto create_entity() -> Entity;

    // Take back the entity to give out later again
    void destroy_entity(Entity entity);

    void set_entity_signature(Entity entity, Signature signature);

    auto get_entity_signature(Entity entity) -> Signature;

private:
    std::queue<Entity> available_entities;
    std::array<Signature, MAX_ENTITIES> entity_signatures;
};
```

ECS - Component Array

For each component, we can store the data in a contiguous array (FAST!)

```
template <typename T>
struct ComponentArray {
    std::array<T, MAX_COMPONENTS> component_array;
};

ComponentArray<Position> position_components;
```

Entities are just indices, but:

- If entities are created and destroyed, some indices may not be valid
- Not all entities have all components

Thus, this array will have gaps if we naively implement this ...

ECS - Component Array

We can use a map to track which Entity to packed index mappings.
Example: Position Component Array

```
Array: [

]
EntityToIndex: [

]
IndexToEntity: [

]
Size: 0
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Add Position {0, 1} to Entity 0

Array: [
    {0, 1}
]
EntityToIndex: [
    {0:0}
]
IndexToEntity: [
    {0:0}
]
Size: 1
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Add Position {1, 2} to Entity 2

Array: [
    {0, 1}, {1, 2}
]
EntityToIndex: [
    {0:0}, {2:1}
]
IndexToEntity: [
    {0:0}, {1:2}
]
Size: 2
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Add Position {3, 0} to Entity 5

Array: [
    {0, 1}, {1, 2}, {3, 0}
]
EntityToIndex: [
    {0:0}, {2:1}, {5:2}
]
IndexToEntity: [
    {0:0}, {1:2}, {2:5}
]
Size: 3
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Add Position {4, 4} to Entity 8

Array: [
    {0, 1}, {1, 2}, {3, 0}, {4, 4}
]
EntityToIndex: [
    {0:0}, {2:1}, {5:2}, {8:3}
]
IndexToEntity: [
    {0:0}, {1:2}, {2:5}, {3:8}
]
Size: 4
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Remove Entity 2 (swap array element to end)

Array: [
    {0, 1}, {1, 2}, {3, 0}, {4, 4}
]
EntityToIndex: [
    {0:0}, {2:1}, {5:2}, {8:3}
]
IndexToEntity: [
    {0:0}, {1:2}, {2:5}, {3:8}
]
Size: 4
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Remove Entity 2 (swap array element to end)
                ----------------
Array: [        |               |
    {0, 1}, {4, 4}, {3, 0}, {1, 2}
]
EntityToIndex: [
    {0:0}, {2:3}, {5:2}, {8:1}
]
IndexToEntity: [
    {0:0}, {1:8}, {2:5}, {3:2}
]
Size: 4
```

ECS - Component Array

We can use a map to track which Entity to packed index mappings:
Example: Position Component Array

```
Remove Entity 2 (swap array element to end)

Array: [
    {0, 1}, {4, 4}, {3, 0}
]
EntityToIndex: [
    {0:0}, {5:2}, {8:1}
]
IndexToEntity: [
    {0:0}, {2:5}, {3:2}
]
Size: 3
```

ECS - Component Array

```cpp
// Will need to call interface on list of multiple tempalted types
struct BaseComponentArray {
    virtual ~BaseComponentArray() = default;
    virtual void entity_destroyed(Entity entity) = 0;
};

template <typename T>
struct ComponentArray : public BaseComponentArray {
    // Insert entity with component data
    void insert(Entity entity, T component);

    // Remove component data from entity
    void remove(Entity entity);

    // Get the entities component data
    auto get_entity_data(Entity entity) -> T&;

    // Signal an entity destroyed and needs to update mappings
    void entity_destroyed(Entity entity) override;

    std::array<T, MAX_ENTITIES> component_array;
    std::unordered_map<Entity, std::size_t> entity_to_index;
    std::unordered_map<std::size_t, Entity> index_to_entity;
};
```

ECS - Component Manager

The `ComponentManager` is responsible managing each of the `ComponentArrays` such as adding or removing a component from an Entity.

Since we are tracking which components belong to which entities using N bits, each component will need a unique ID which maps to the bit index.

```
class component_type_counter {
    static inline ComponentType current_index = 0;

public:
    template <typename T>
    static inline const ComponentType index_of = current_index++;
};
```

ECS - Component Manager

```
struct ComponentManager {
    template <typename T>
    void register_component();

    // Same index into bitset
    template <typename T>
    auto get_component_type() -> ComponentType;

    // Add component to entity
    template <typename T>
    void add_component(Entity entity, T component);

    // Remove component from entity
    template <typename T>
    void remove_component(Entity entity);

    // Notify each component array than an entity has been destroyed
    void entity_destroyed(Entity);

    std::unordered_map<ComponentType, std::shared_ptr<BaseComponentArray>>
    ↪  component_arrays;
};
```

ECS - System

A System implements some functionality upon a lit of entities with a certain signature of components.

```cpp
// Forward declare to avoid recursive includes
// Acts as a coordinator between Entities, Components, and Systems
//  to pass information between
class Coordinator;

// Iterated over a collection of entities of a particular signature
// Meant to be subclassed
class System {
public:
    virtual ~System() = default;

    // Update the system after step_seconds has passed since the past update
    virtual void update(Coordinator &world, double step_seconds);

    // What entities belong to the system
    std::set<Entity> entities;
};
```

ECS - System Example

Consider the collision handler from Assignment 2. We can create a `System` which implements that behaviour on the set of entities which have a `CollisionComponent`.

```
class CollisionSystem : public System {
public:
    void update(Coordinator &world, double step_seconds) override {
        for (const auto &entity : entities) {
            auto &collisions =
                world.get_component<CollisionComponent>(entity);
            auto &heading = world.get_component<HeadingComponent>(entity);
            if (collisions[LEFT] || collisions[RIGHT]) {
                heading.x = -heading.x;
            }
            if (collisions[TOP] || collisions[BOTTOM]) {
                heading.y = -heading.y;
            }
            collisions.reset();
        }
    }
};
```

ECS - System Manager

The `SystemManager` is responsible managing each system.

Again, we use the same *trick* to track system types to indices in our data structures:

```
class system_type_counter {
    static inline std::size_t current_index = 0;

public:
    template <typename T>
    static inline const std::size_t index_of = current_index++;
};
```

ECS - System Manager

```
class SystemManager {
    // Register (create) a new system constructing with the passed args
    // The priority lets us control which systems are updated in what order
    template <typename T, typename ...Args>
    auto register_sytem(int priority, Args ... args) -> std::shared_ptr<T>;

    // Set the signature for collection of components the system T
    //  will operate on
    template <typename T>
    void set_signature(Signature signature);

    // Remove entity from all systems
    void entity_destroyed(Entity entity);

    // Update entity signature for all systems
    void entity_signature_changed(Entity entity, Signature signature);

    // Call the update method for each system
    void update_all(Coordinator &world, double step_seconds);

    std::unordered_map<std::size_t, Signature> signatures;
    std::unordered_map<std::size_t, std::shared_ptr<System>> systems;
    std::map<int, std::vector<std::shared_ptr<System>>> systems_by_priority;
};
```

ECS - Coordinator

The `Coordinator` is a single object which we will use to interface between the entities, components, and systems.

```cpp
class Coordinator {
public:
    Coordinator()
        : component_manager(std::make_unique<ComponentManager>()),
          entity_manager(std::make_unique<EntityManager>()),
          system_manager(std::make_unique<SystemManager>()) {}
    // ...
private:
    std::unique_ptr<ComponentManager> component_manager;
    std::unique_ptr<EntityManager> entity_manager;
    std::unique_ptr<SystemManager> system_manager;
    std::unordered_map<std::string, std::any> data;
};
```

ECS - Coordinator

Entity methods:

```
class Coordinator {
public:
    auto create_entity() -> Entity {
        return entity_manager->create_entity();
    }

    void destroy_entity(Entity entity) {
        entity_manager->destroy_entity(entity);
        component_manager->entity_destroyed(entity);
        system_manager->entity_dstroyed(entity);
    }
};

Entity player1 = coordinator.create_entity();
coordinator.destroy_entity(player1);
```

ECS - Coordinator

Component methods:

```
class Coordinator {
public:
    template <typename T>
    void register_component() {
        component_manager->register_component<T>();
    }

    template <typename T> // Add component to an entity
    void add_component(Entity entity, T component);

    template <typename T>  // Remove component from an entity
    void remove_component(Entity entity);

    template <typename T>  // Get component data for the entity
    auto get_component(Entity entity) -> T &;

    template <typename T>
    auto get_component_type() -> ComponentType;
};

coordinator.register_component<Position>()
coordinator.add_component<Position>(player1, {10, 20});
```

ECS - Coordinator

System methods:

```
class Coordinator {
public:
    // System methods
    template <typename T, typename... Args>
    auto register_system(int priority, Args... args) -> std::shared_ptr<T> {
        return system_manager->register_system<T>(priority, args...);
    }

    template <typename T>
    void set_system_signature(Signature signature);

    void update_all_standard_systems(double step_seconds);

    void update_all_render_systems(double step_seconds);

    // Misc
    void store_data(const std::string &name, std::any value);

    template <typename T>
    auto get_data(const std::string &name) -> T &;
};
```

Game Loop

All game loops look something like the following:

```
while (true) {
    process_input();
    update_world();
    render();
}
```

Every loop iteration (*tick*), we process input from users, update entities in the world, then render the current state of the world.

Unfortunately, we have no control over the speed at which the game progresses:

- *Light weight* games will run too fast for the users to interpret what is going on.

Game Loop - Target FPS

We can modify the previous game loop by using a target frames-per-second (FPS). If we can process input, update the game state, and render in under the FPS target, our game will run at a smooth consistent framerate:

```cpp
const double FPS = 60;
const double MS_PER_SECOND = 1000;
const double MS_PER_FRAME = MS_PER_SECOND / FPS;

while (true) {
    double start_time = get_current_time();
    process_input();
    update_world();
    render();
    double end_time = get_current_time();
    double sleep_time = (start_time - end_time) + MS_PER_FRAME;
    std::this_thread::sleep_for(sleep_time);
}
```

What happens if we cannot process the game state fast enough for our target FPS? Are there any other downsides?

Game Loop - Syncing The Clocks

There are two clocks we need to be aware of:

1. Each world update advances the game clock by a certain amount
2. It takes a certain amount of *real* time to process that

If step 2 takes longer than step 1 (i.e. it takes more than 16ms of processing to advance the game world clock by 16ms) then the game slows down.

If we can update the game by *more* than 16ms of game time in a single step, then we can update less frequently and still keepup:

- Choose a time step to advance based on how much *real* time has passed since the last frame
- Longer frame times results in bigger steps taken when updating the game forward
- We always stay in sync with real time, as we change our world step size based on the real clock

Game Loop - Syncing The Clocks

Choose a time step for the game world to update that matches how long its taken for the real clock to process the last step (i.e. always play catchup)

```
double prev_time = get_current_time();

while (true) {
    double curr_time = get_current_time();
    double elapsed_time = curr_time - prev_time;
    process_input();
    update_world(elapsed_time);
    render();
    prev_time = curr_time;
}
```

Game Loop - Syncing The Clocks

Pros:

- Game plays at consistent rate on different hardware
- Faster hardware will result in more engine updates (with smaller step sizes), but also with more render steps (i.e. higher frame rate)

Cons:

- Engine is non-deterministic (update step size is determined by hardware speed)!
- Many smaller game step updates can result in math stability issues:
  - Floating point math has rounding errors and will accumulate with more updates
  - More updates with smaller delta values can lead to more rounding errors

Game Loop - Playing Catchup

Ideally, we should use a fixed timestep to update the game state.

- Deterministic game engine
- Can control amount of accumulation errors

Once insight is that rendering is not affected by our time steps:

- The renderer captures an instant in time and draws that on screen
- It doesn't matter how much time has advanced in the engine, it just grabs the current state of the world

Solution:

- Update the game engine at fixed intervals of time (game engine tick rate)
- Continue to update until we catch up to the amount of *real time* has passed since the last frame
- Once we are caught up, draw the frame

Game Loop - Playing Catchup

We use the variable `lag` to represent how much *real* time has passed since the last frame. This measures how much time the engine is behind the real world clock.

The inner loop then runs until we have *caught up*:

```
double prev_time = get_current_time();
double lag = 0;

while (true) {
    double curr_time = get_current_time();
    double elapsed_time = curr_time - prev_time;
    prev_time = curr_time;
    lag += elapsed;
    process_input();

    while (lag >= MS_PER_UPDATE) {
        update_world(elapsed_time);
        lag -= MS_PER_UPDATE;
    }
    render();
}
```

References

[1] A. Morlan, "A Simple Entity Component System (SCS)"
`http://austinmorlan.com/posts/entity_component_system`

[2] R. Nystrom, "Game Programming Patterns", Genever Benning, 201