Entity Component System (ECS)

ECS is used in Minecraft and many other commercial games
What are Entities?

- **Entities**: things that exist in your game world
Entities in Traditional Game Programming

• Object-Oriented Programming
  • **Entities as objects**
    • Contains data, behaviors, etc.
  • **Entity Hierarchy: Entities extend other Entities**
Entity Hierarchy (object oriented design)

class Entity {
public:
    void create();
    void destroy();
    void move();
private:
    double x;
    double y;
    double vel_x;
    double vel_y;
    vec2 bbox;
    float health;
    GLuint texture;
}

class Player : public Entity {
public:
    void jump();
    bool collide();
}

class Mario : public Player {
public:
    void power_up();
}

class Enemy : public Entity {
private:
    bool squishable;
}

class Goomba : public Goomba {
public:
    void squish();
}
Issues with Object-Oriented Approach

What if we want Mario to be able to be squished?
Issues with Object-Oriented Approach

- Difficult to add **new** behaviors
- **Choice between replicating code or**
- **MONSTER SIZE parent classes**

Both options aren’t ideal for big games!
Goomba is now separated from its data & methods

Example ECS Diagram

Goomba
- Sprite Component
- Position Component
- Velocity Component
- Physics Component
- Collision Component

int index
Now what if we want Mario to be able to be squished?
We can give Mario a Physics Component to make him squishable.
What would happen to Mario here?

Example ECS Diagram

Mario
int index

Sprite Component
GLuint texture

Position Component
double x
double y

Velocity Component
double vel_x
double vel_y

Physics Component
bool squishable
vec2 bbox

Collision Component

Render System
void draw()

Motion System
void move()

Physics System
void squish()
What is ECS?

- Alternative to object-oriented programming
- Data is self-contained & modular
  - Similar concept to building blocks
  - Entities no longer “own” data
  - Entities pick & choose
What is ECS?

• Entities actions determined only by their data
  • *Update loop doesn’t need references to Entities*
  • *Systems search for Entities with right parts (data) & update*
  • For Mario to move he needs a position & velocity
What is ECS?

- **Composition over hierarchy**
- **Entities** are collections of **Components**
- **Components** contain *game data*
  - *Position, velocity, input, etc.*
- **Systems** are collections of **actions**
  - *Render system, motion system, etc.*
Component

- Contains **only** game data
- Describes **one** aspect of an Entity
  - *ex. a trumpet Entity will likely have an audio Component*

<table>
<thead>
<tr>
<th>Sprite Component</th>
<th>Position Component</th>
<th>Velocity Component</th>
<th>Physics Component</th>
<th>Collision Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLuint texture</td>
<td>double x</td>
<td>double vel_x</td>
<td>bool squishable</td>
<td>vec2 bounding_box</td>
</tr>
<tr>
<td></td>
<td>double y</td>
<td>double vel_y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input Component</th>
<th>AI Component</th>
<th>Health Component</th>
<th>Audio Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool left</td>
<td>bool do_left</td>
<td>float health</td>
<td>mp3 sound</td>
</tr>
<tr>
<td>bool right</td>
<td>bool do_right</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool jump</td>
<td>bool do_jump</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool attack</td>
<td>bool do_shoot</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Component

- Typically implemented with structs.

```c
struct SpriteComponent {
    GLuint texture;
}

struct PositionComponent {
    double x;
    double y;
}

struct VelocityComponent {
    double vel_x;
    double vel_y;
}

struct PhysicsComponent {
    bool squishable;
}

struct CollisionComponent {
    vec2 bbox;
}
```
What Components to Make?

- What Components would we give to the following Entities?
Components

- Easy to add new Entity characteristics
  - *Just create the desired Component & give to Entity*

How do we change our playable hero from Mario to Luigi?
Components

- Empty Components can be used to tag Entities

Empty components are useful, a flag indicating an ability!
## Components

- Empty Components can be used to tag Entities

**Input Component**
- bool left
- bool right
- bool jump
- bool attack

**Sprite Component**
- GLuint texture

**Player Component**

<table>
<thead>
<tr>
<th>Position Component</th>
<th>Velocity Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>double x</td>
<td>double vel_x</td>
</tr>
<tr>
<td>double y</td>
<td>double vel_y</td>
</tr>
</tbody>
</table>

**Sprite Component**

<table>
<thead>
<tr>
<th>Position Component</th>
<th>Velocity Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>double x</td>
<td>double vel_x</td>
</tr>
<tr>
<td>double y</td>
<td>double vel_y</td>
</tr>
</tbody>
</table>

Now Luigi can be identified as the active player
Systems

- Groups of Components describe behavior/action
  - *ex. bounding box, position & velocity describe collisions*

- Systems code behaviors/actions

- Operate on Entities with related groups of components
  - *Related: describe same (type of) behavior/action*
  - *ex. render all Entities with sprite & position*

- Entity behavior can be dynamic
  - *Add/remove components on the fly*
**System Example**

- **What systems might these related groups of components describe?**

<table>
<thead>
<tr>
<th>Position Component</th>
<th>Velocity Component</th>
<th>AI Component</th>
<th>Player Component</th>
<th>Input Component</th>
<th>Position Component</th>
<th>Velocity Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>double x</td>
<td>double vel_x</td>
<td>bool do_left</td>
<td>bool left</td>
<td></td>
<td></td>
<td>double vel_x</td>
</tr>
<tr>
<td>double y</td>
<td>double vel_y</td>
<td>bool do_right</td>
<td>bool right</td>
<td></td>
<td></td>
<td>double vel_y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bool do_jump</td>
<td>bool jump</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bool do_shoot</td>
<td>bool attack</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Alla Sheffer, Helge Rhodin
System Example

- What systems might these related groups of components describe?

<table>
<thead>
<tr>
<th>Position Component</th>
<th>AI Component</th>
<th>Player Component</th>
<th>Input Component</th>
<th>Position Component</th>
<th>Velocity Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>double x</td>
<td>bool do_left</td>
<td>double x</td>
<td>bool left</td>
<td>double x</td>
<td>double vel_x</td>
</tr>
<tr>
<td>double y</td>
<td>bool do_right</td>
<td>double y</td>
<td>bool right</td>
<td>double y</td>
<td>double vel_y</td>
</tr>
<tr>
<td>Velocity Component</td>
<td>bool do_jump</td>
<td>bool jump</td>
<td>bool attack</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double vel_x</td>
<td>double do_shoot</td>
<td>double vel_y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Enemy Motion System
Player Motion System
System Examples

Physics System  ... iterates over all components of type velocity

```cpp
for (Velocity& velocity : velocity_components)
    velocity += 9.81 * dt
```

Game loop

```cpp
Entity player;
if (!alive_entities.has(player)) exit();
```

Motion System  ... iterates over all entities that have velocity and position

```cpp
for (int entity : velocity_entities)
    if (position_entities.has(entity))
        position_components.get(entity) += velocity_components.get(entity);
```
ECS implementations
Memory & ECS

Where do we store our Components?

• RAM, harddrive, or chache?

• Inside Systems?
  • Better, but could be improved
  • Different Systems may need the same Component types
    • How do we decide who owns what?
    • Messaging can get overly complex between systems
### Problem: associating entities and components

<table>
<thead>
<tr>
<th>Position</th>
<th>Velocity</th>
<th>Jumps</th>
<th>Player</th>
<th>Squishable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Object-oriented-programming (OOP)?

ECS = containers of components?
Memory & ECS

Where do we store our Components?

• Inside Entities?

Update loop has to access non-contiguous memory repeatedly!

Slow memory access!

Memory Blocks

- position
- velocity
- collision
- sprite
The Map Approach
(entity ID to component address)

Concept: A (hierarchical) acceleration structure to lookup components
Implementation: std::map<Entity, Position>
Memory & ECS

Where do we store our Components?

• In a map?

Update loop has to access non-contiguous memory repeatedly!

Slow memory access!

Memory Blocks
The (giant) Sparse Array

<table>
<thead>
<tr>
<th>ID</th>
<th>Position</th>
<th>Velocity</th>
<th>Jumps</th>
<th>Player</th>
<th>Squishable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Concept:** A huge data matrix of size Nr. Entities x Nr. components

**Implementation:** std::vector<Position>; std::vector<Velocity>
Memory & ECS

Where do we store our Components?

- Array with holes?

Better cache utilization!

Not memory efficient!

Memory Blocks

- position
- velocity
- collision
- sprite
**Bitset / Bitmap**

<table>
<thead>
<tr>
<th>ID</th>
<th>Bitset/Bitmap</th>
<th>Position</th>
<th>Velocity</th>
<th>Jumps</th>
<th>Player</th>
<th>Squishable</th>
<th>Issues?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td>1 11110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td>2 11001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Concept:** Each entity has a bitset that is true for its ‘owned’ components

**Implementation:** long bitset; // how many components can we support? If(bitset & query == query) // has the entity all query components?
Key & Lock Metaphor

Systems will only operate on Entities with the required Components

Motion System
Further Improvements
Dense Component Vectors
(an attempt, needs more)

Concept: One array/vector per component, but how to associate?
Implementation: std::vector<Position>; std::vector<Velocity> + X?

<table>
<thead>
<tr>
<th>ID</th>
<th>Position</th>
<th>Velocity</th>
<th>Jumps</th>
<th>Player</th>
<th>Squishable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How to find the position of Goomba’s squishable component?

Issues?
Map + Dense Component Vectors (entity ID to component address index)

Concept: Combine dense vectors with a map
Implementation: std::vector<Component>; std::map<Entity,unsigned int>

Issues?
Map + Dense Vector (different visualization)
Cache is Key

- Each Component type has a **statically** allocated array
- Minimizes costly cache misses
  - *Keeps components we access around the same time close to each other*
Map + Component Vector + Entity Vector

Concept: Add a dense vector of entities to facilitate quick iteration over entities
Implementation: std::vector<Entities>; std::vector<Component>; std::map<Entity,unsigned int>

Easy to iterate over all velocity components that belong to an entity with a position

```cpp
for(int entity : velocity_entities) // using the key array
    if (position_entity_map.has(entity)) // using the map
        position_entity_map.get(entity) += velocity_entity_map.get(entity); // using component array
```
Faster iteration via entity and component array

Accessing the velocity map (reg_velocity.map) is an unnecessary indirection

```java
for(int entity : velocity_entities) // efficient
    if (position_entity_map.has(entity)) // inefficient lookup
        position_entity_map.get(entity) += velocity_entity_map.get(entity); // 2x inefficient lookup
```

We can access the velocity components in linear fashion

```java
for(int vel_i = 0; vel_i < velocity_entities.size(); vel_i++) // efficient
    Entity entity : velocity_entities[vel_i]; // efficient
    int pos_i = position_entity_map.getIndex(entity); // inefficient lookup
    if (pos_i)
        position_components[pos_i] += reg_velocity_components[vel_i]; // efficient
```
Map + Component Vectors + Entity Vector Cache is Key

Map access slow

Update loop accesses contiguous memory **IDEAL!**

- position
- velocity
- collision
- sprite
- position entities
- velocity entities
- collision entities
- sprite entities
**Advanced ECS: Archetypes / prototypes / pools**

- **Concept:** store all types with the same components in dense arrays
- Used by the Unity ECS system
- Difficult to implement

<table>
<thead>
<tr>
<th></th>
<th>Goomba1</th>
<th>Goomba2</th>
<th>Luigi</th>
<th>Mario</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Velocity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Squishable</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. **Goomba1**
2. **Goomba2**
3. **Luigi**
4. **Mario**

© Alla Sheffer, Helge Rhodin
How Does a System Find its Entities?

Extension: Entity Manager

• Each system has a list of entity IDs it is interested in
• Systems register their bitsets/bitmaps with the Entity Manager
• Whenever an Entity is added…
  – Evaluate which systems are interested & update their ID lists
Self-study: A special map approach
### Self-study: The ‘Sparse Set’

**Concept:** Sparse array + dense array

**Implementation:** std::vector<Entity> entities; std::vector<unsigned int> indices; std::vector<Components> components;

<table>
<thead>
<tr>
<th>ID</th>
<th>Position</th>
<th>Velocity</th>
<th>Jumps</th>
<th>Player</th>
<th>Squishable</th>
<th>Issues?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Index Pos: 1
2. Index Vel: 1
3. Index Jump: 2
4. Index Player: 2
5. Index Squishable: 0

<table>
<thead>
<tr>
<th>ID</th>
<th>Index Pos</th>
<th>Index Vel</th>
<th>Index Jump</th>
<th>Index Player</th>
<th>Index Squishable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mario</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Luigi</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Goomba2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Alla Sheffer, Helge Rhodin
Self-study: Faster Lookup with Sparse Sets

The map lookup (map.get(entity)) is costly
• A hashmap is O(1), but that 1 is big

Sparse set:
• An array as large as the number of entities in the game
  • Crazy waste of memory?!
  • 32 bit integer -> ???
  • a sparsely filled array
• A small dense array of all entities in sequence (as before)
  • Extremely fast lookup, insert, & clear
Entity Summary

- Each Entity is typically just a unique identifier to its components
- Store Entities in a big static array in the Entity Manager
  - *Monitor removed entities*
Memory & ECS

Where do we store our Components?

• Inside a registry!
  • Systems don’t own components
  • One big array for each Component type
  • Takes advantage of modular architecture of ECS

YES!
Cache is Key

• When we “delete” an entity we must delete corresponding components to.

• Different approaches to this,
  – Fill deleted components in arrays with the last entities data
    ▸ Extra care must be taken when managing indices
  – Mark spots in arrays as rewritable
    ▸ Big systems will suffer from poor memory management
Entity Component Systems: Benefits

- **Complexity**
  - *Game code tends to grow exponentially*
  - *Complexity of ECS architecture does not grow with it*
  - *Easy to maintain*

- **Customization**
  - *Games have a lot of dynamic operations*
  - *Add/remove components to change Entity behavior*
  - *ECS is highly modular*

- **Can be very memory efficient!**
The game loop

Can you imagine a game without?
A game is a simulator

1. AI and user input
2. Environment reaction
3. Equations of Motion
   - sum forces & torques, solve for accelerations: $\vec{F} = ma$
4. Numerical integration
   - update positions, velocities
5. Collision detection
6. Collision resolution

$\leftarrow$ Also simulation forms!

We will have a separate lecture on physics simulation!
// Set all states to default
world.restart();
auto t = Clock::now();
// Variable timestep loop
while (!world.is_over())
{
    // Processes system messages, if this wasn't present the window would become unresponsive
    glfwPollEvents();

    // Calculating elapsed times in milliseconds from the previous iteration
    auto now = Clock::now();
    float elapsed_ms = static_cast<float>((std::chrono::duration_cast<std::chrono::microseconds>(now - t)).count()) / 1000.f;
    t = now;

    DebugSystem::clearDebugComponents();
    ai.step(elapsed_ms, window_size_in_game_units);
    world.step(elapsed_ms, window_size_in_game_units);
    physics.step(elapsed_ms, window_size_in_game_units);
    world.handle_collisions();

    renderer.draw(window_size_in_game_units);
}

return EXIT_SUCCESS;
Backup