

CPSC 427

Video Game Programming

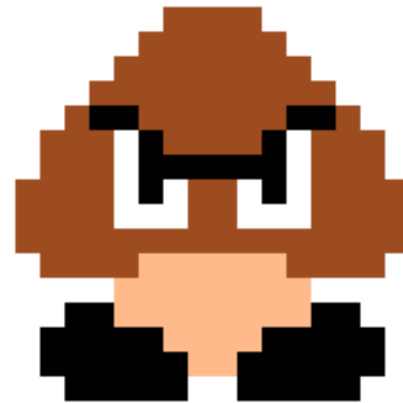
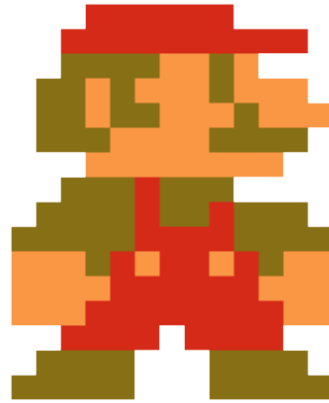
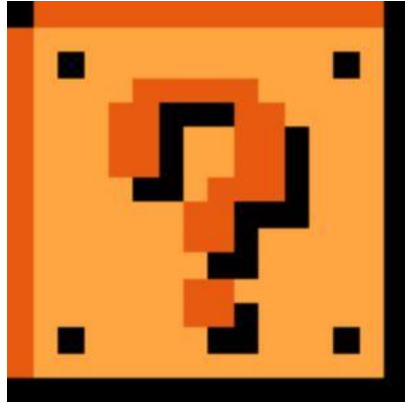
Entity Component System (ECS)



ECS is used in Minecraft and many other commercial games

What are Entities?

- **Entities:** things that exist in your game world



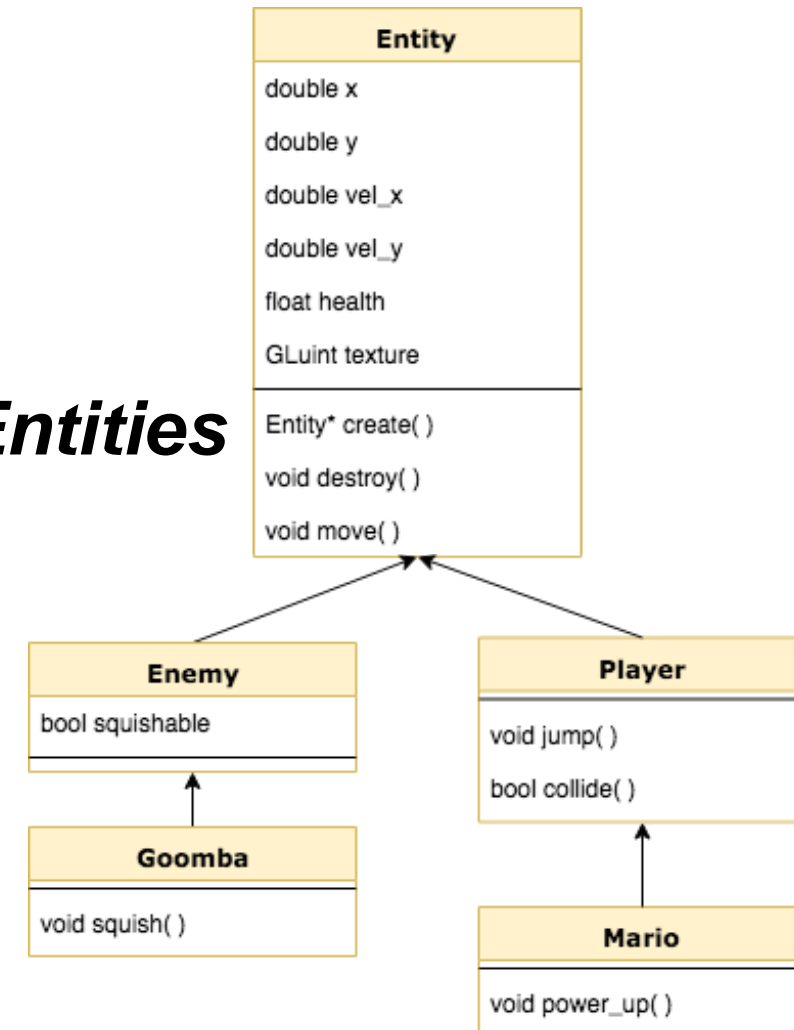
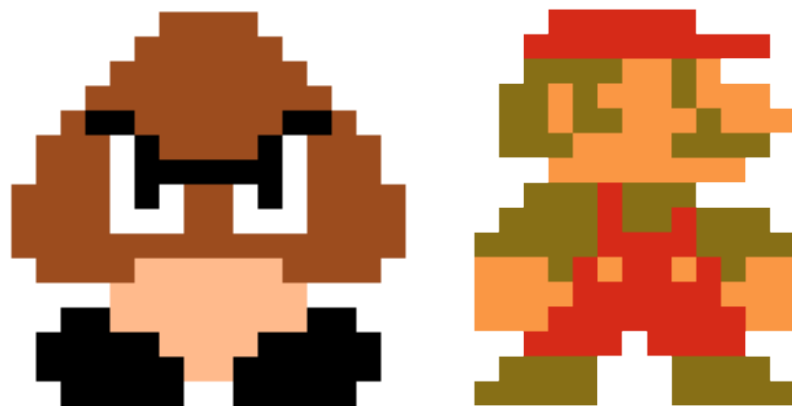
Entities in Traditional Game Programming

- **Object-Oriented Programming**

- *Entities as objects*

- Contains data, behaviors, etc.

- *Entity Hierarchy: Entities extend other Entities*

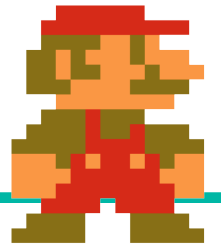


Entity Hierarchy (object oriented design)

```
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();  
}
```



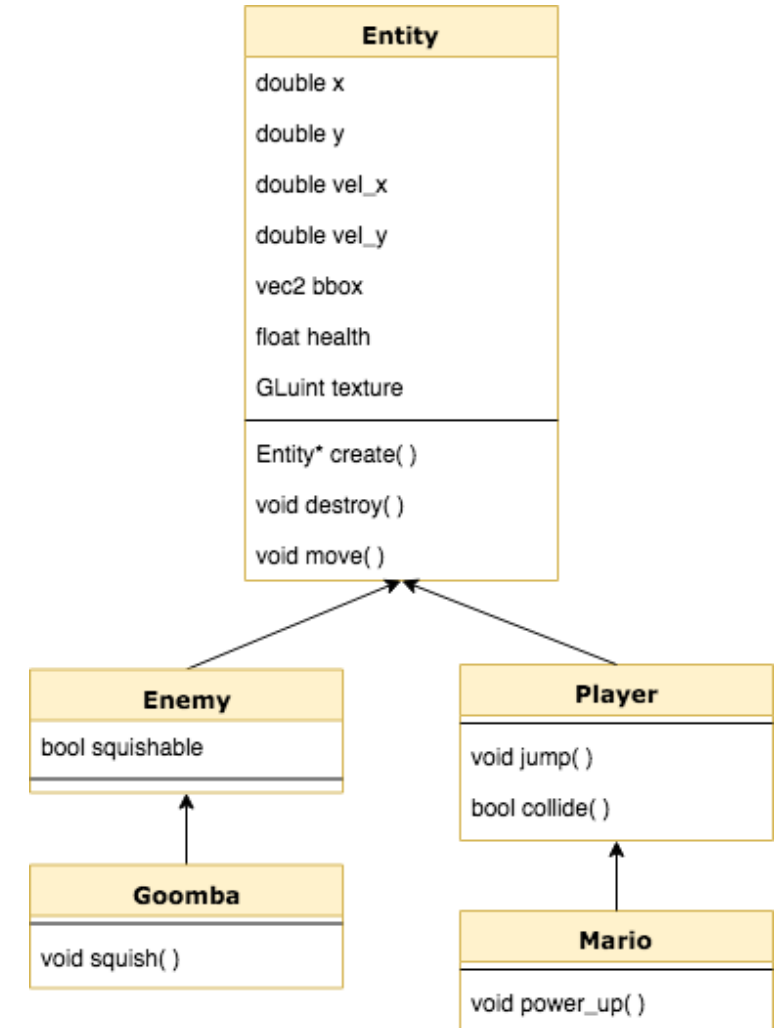
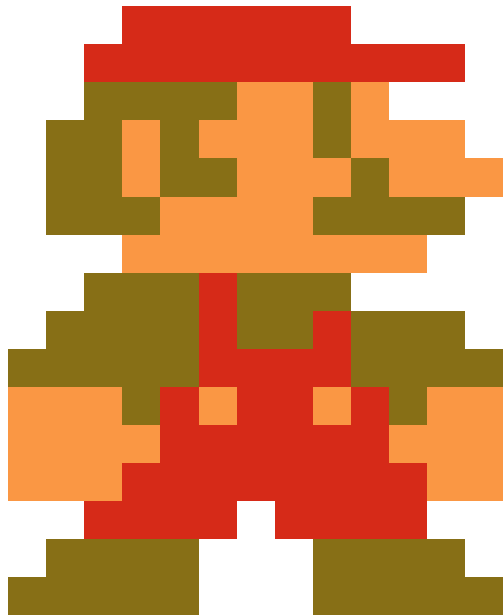
```
class Enemy : public Entity {  
  
private:  
    bool squishable;  
}
```

```
class Goomba : public Goomba {  
  
public:  
    void squish();  
}
```



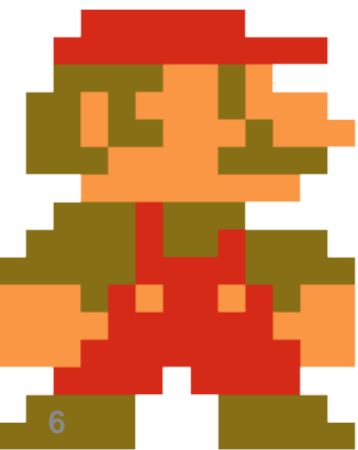
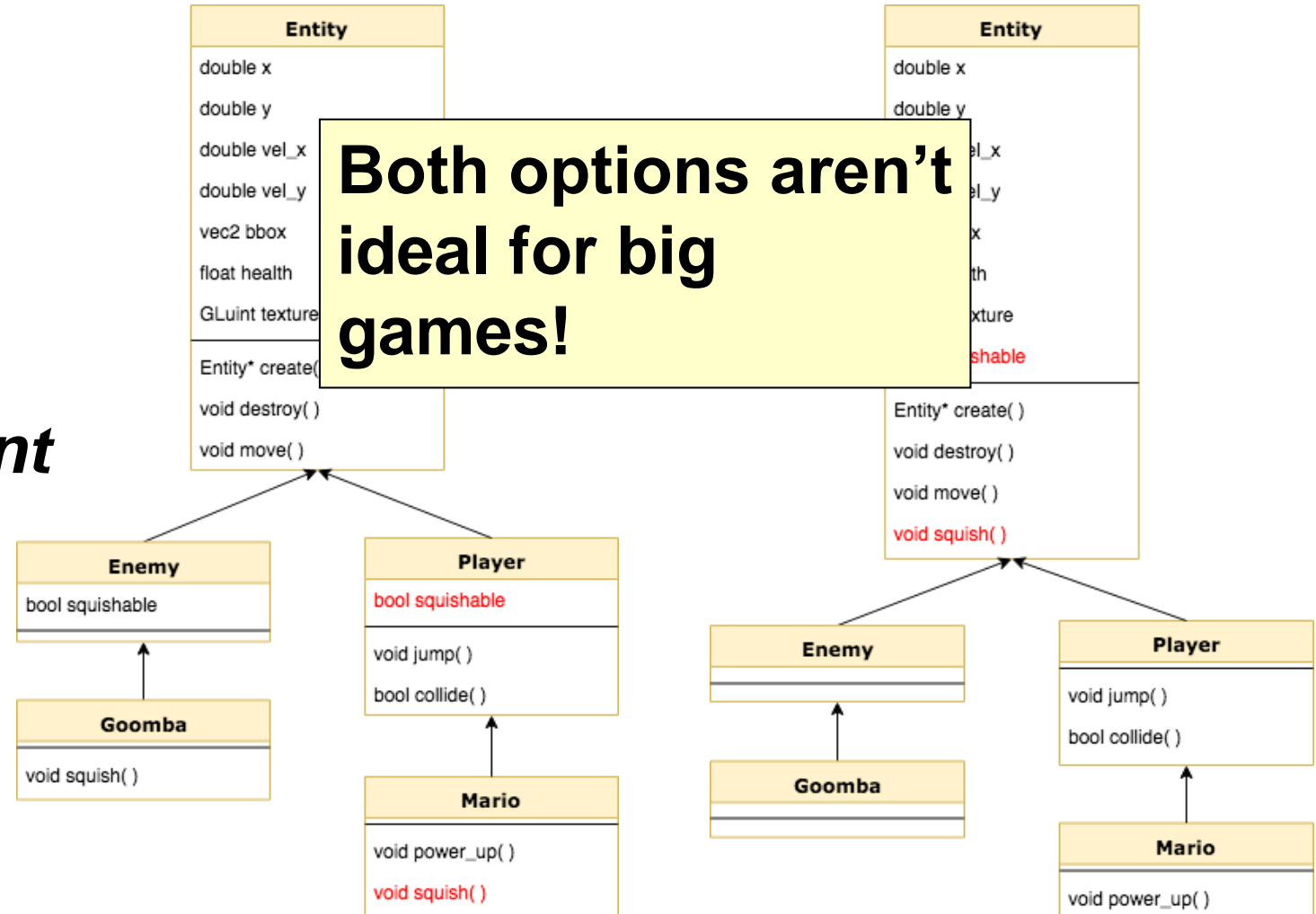
Issues with Object-Oriented Approach

What if we want Mario to be able to be squished?



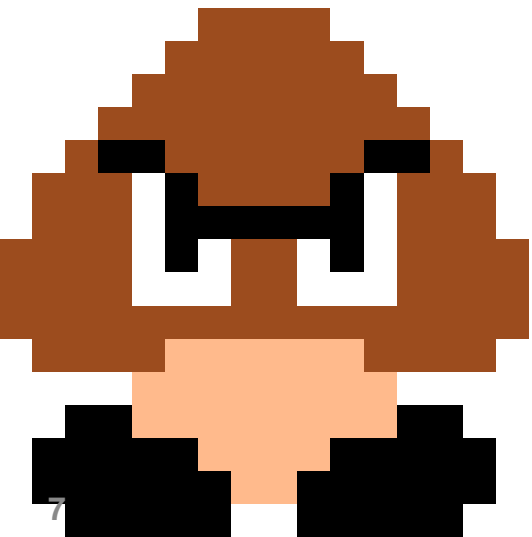
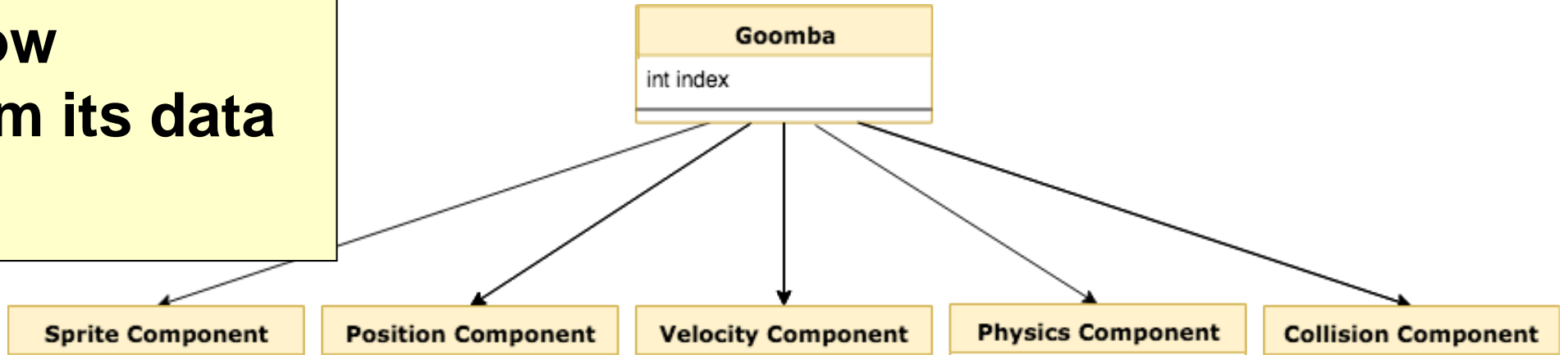
Issues with Object-Oriented Approach

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



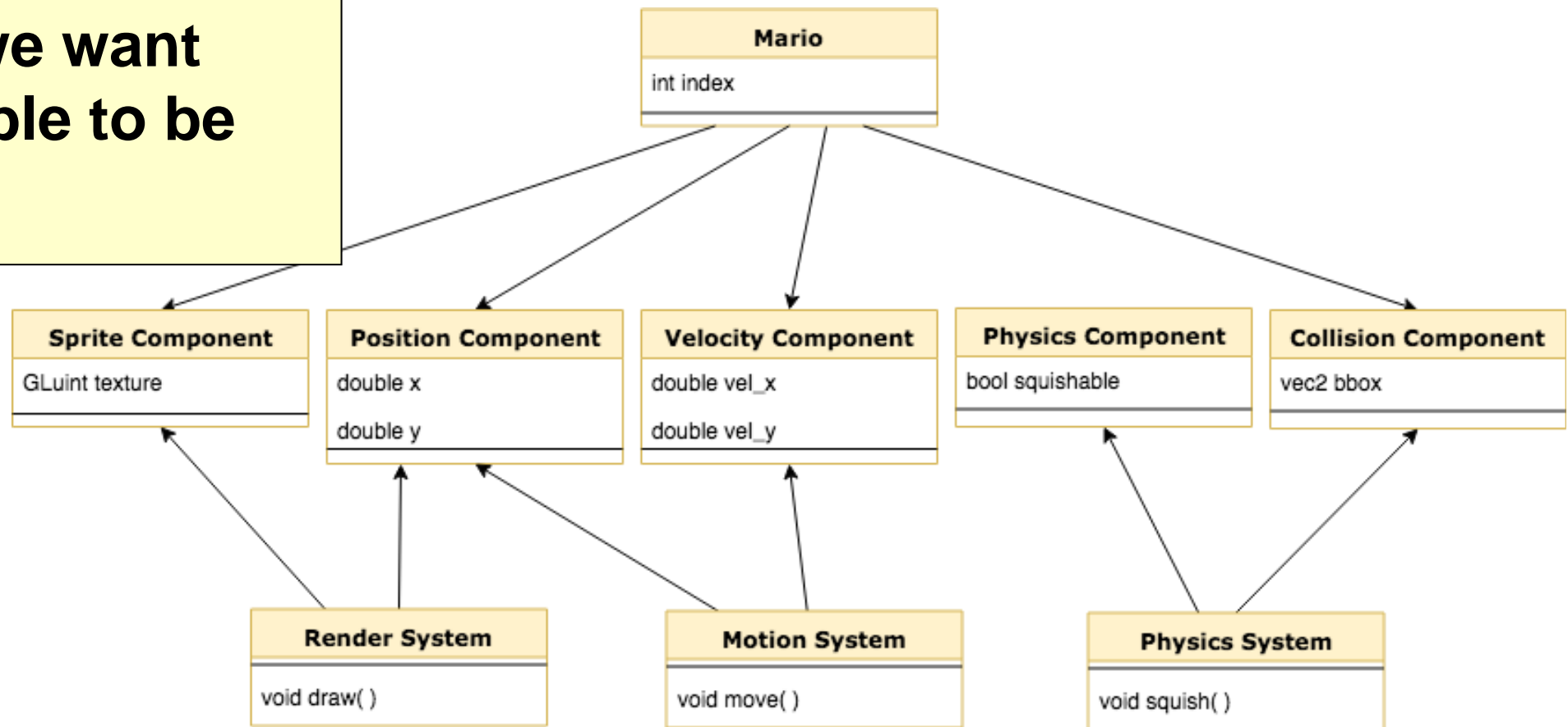
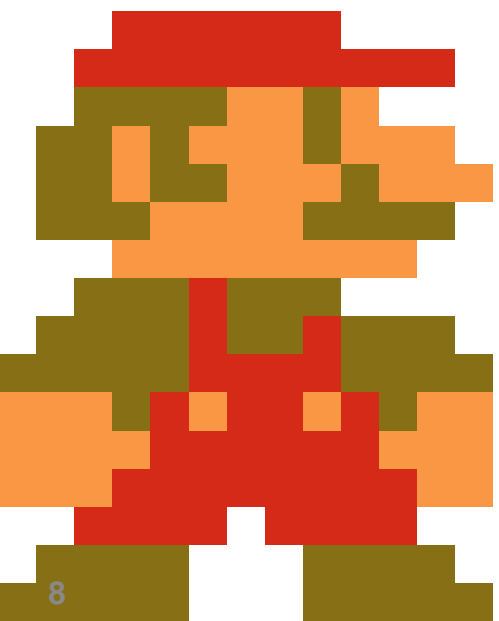
Example ECS Diagram

Goomba is now separated from its data & methods



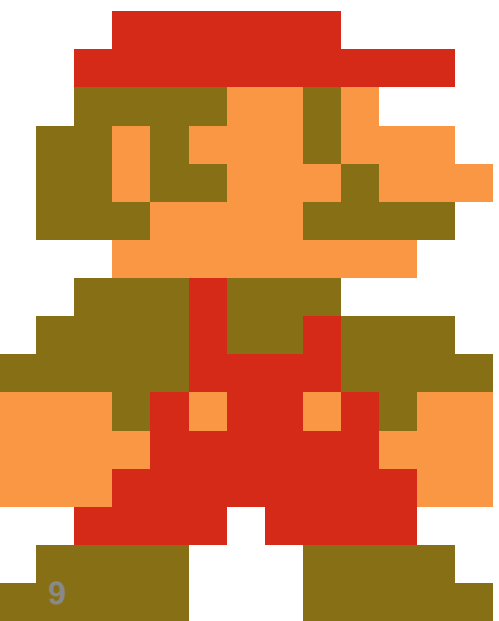
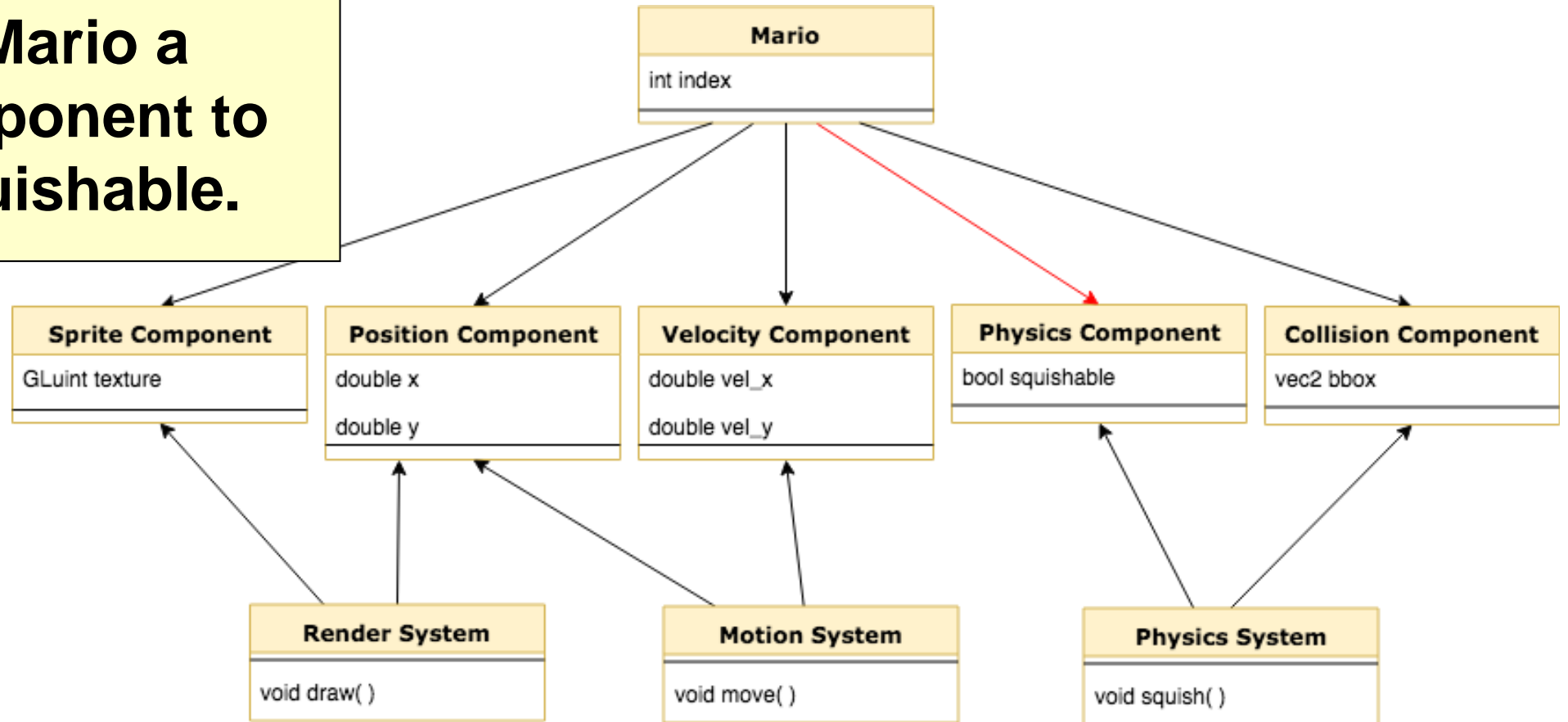
Example ECS Diagram

Now what if we want Mario to be able to be squished?



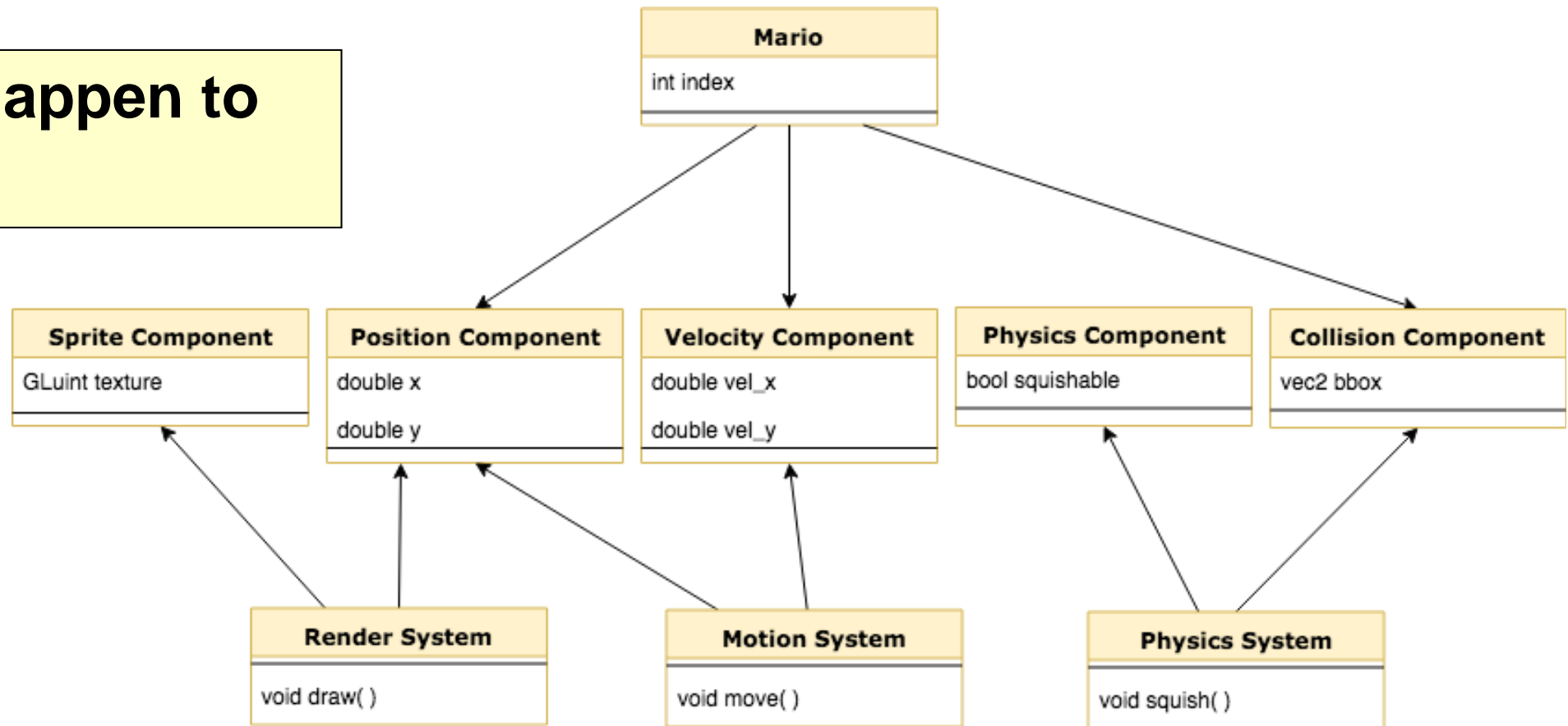
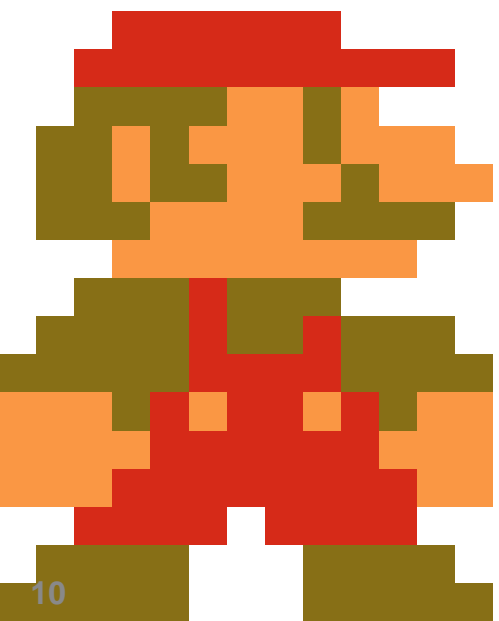
Example ECS Diagram

We can give Mario a Physics Component to make him squishable.



Example ECS Diagram

What would happen to Mario here?



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*

Sprite Component GLuint texture	Position Component double x double y	Velocity Component double vel_x double vel_y	Physics Component bool squishable	Collision Component vec2 bounding_box
Input Component bool left bool right bool jump bool attack	AI Component bool do_left bool do_right bool do_jump bool do_shoot	Health Component float health	Audio Component mp3 sound	

Component

- Typically implemented with structs.

```
struct SpriteComponent {  
    GLuint texture;  
}
```

```
struct PositionComponent {  
    double x;  
    double y;  
}
```

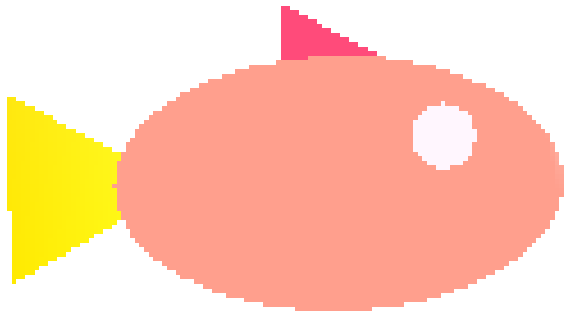
```
struct VelocityComponent {  
    double vel_x;  
    double vel_y;  
}
```

```
struct PhysicsComponent {  
    bool squishable;  
}
```

```
struct CollisionComponent {  
    vec2 bbox;  
}
```

What Components to Make?

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



Components

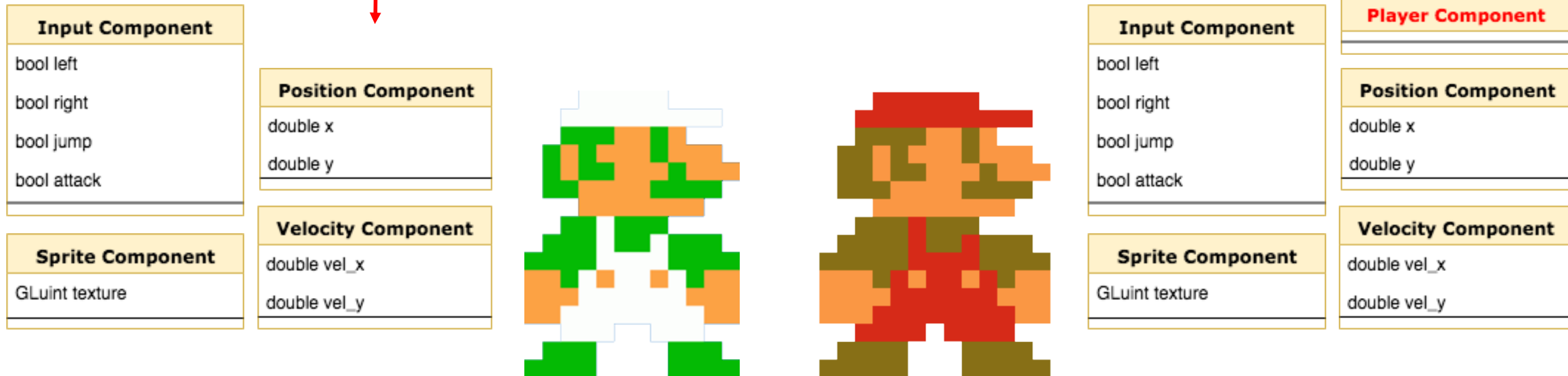
- Easy to add new Entity characteristics
 - *Just create the desired Component & give to Entity*



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

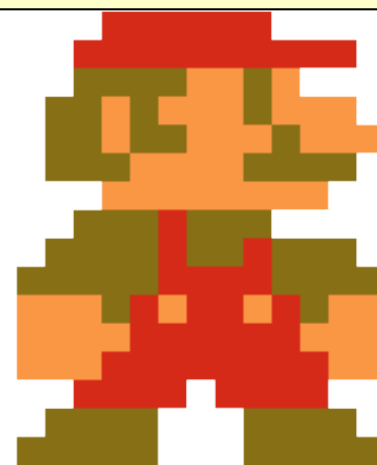
Player Component

Position Component
double x
double y

Velocity Component
double vel_x
double vel_y

Sprite Component
GLuint texture

Now Luigi can be identified as the active player



Entity Component

Sprite Component
GLuint texture

Position Component
double x
double y

Velocity Component
double vel_x
double vel_y

bool attack

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?

Position Component
double x
double y

Velocity Component
double vel_x
double vel_y

AI Component
bool do_left
bool do_right
bool do_jump
bool do_shoot

Player Component

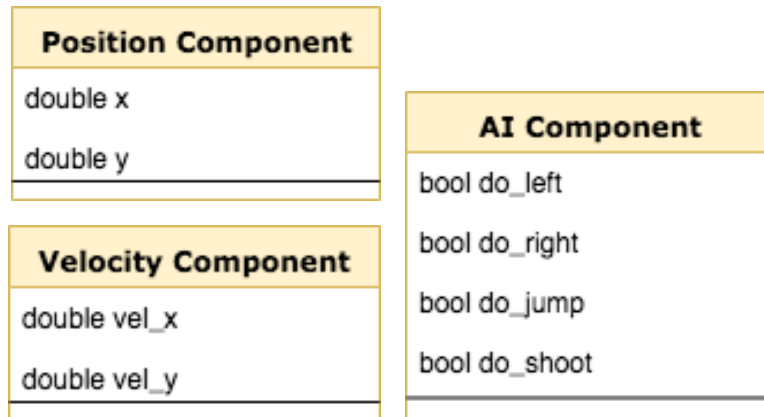
Input Component
bool left
bool right
bool jump
bool attack

Position Component
double x
double y

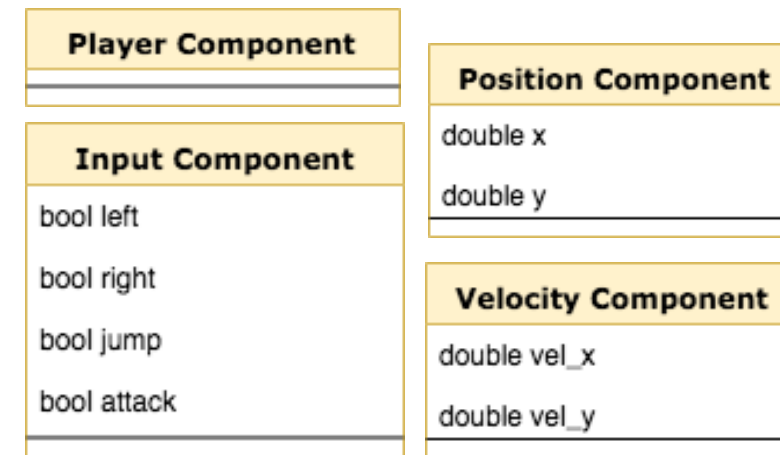
Velocity Component
double vel_x
double vel_y

System Example

- What systems might these related groups of components describe?



Enemy Motion System



Player Motion System

System Examples

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

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

The physics system does not care about entities at all!

Game loop

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

Single boolean check

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

```
for(int entity : velocity_entities)  
    if (position_entities.has(entity))  
        position_components.get(entity) += velocity_components.get(entity);
```

*Need to know all entities that have component X
Need to retrieve a component X from an entity*



ECS implementations

Memory & ECS

Where do we store our Components?

- RAM, harddrive, or cache?
- 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

	Position	Velocity	Jumps	Player	Squishable
Mario	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Goomba1	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>
Luigi	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Goomba2	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>

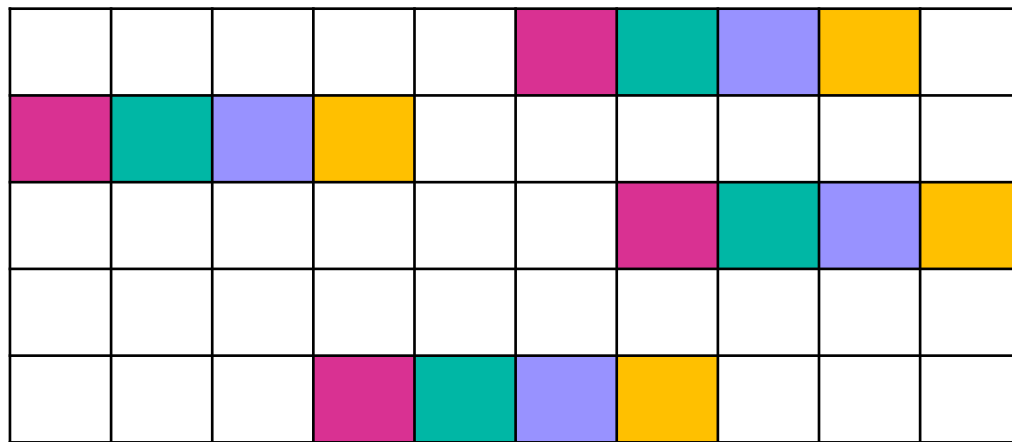
Object-oriented-programming (OOP)?

ECS = containers of components?

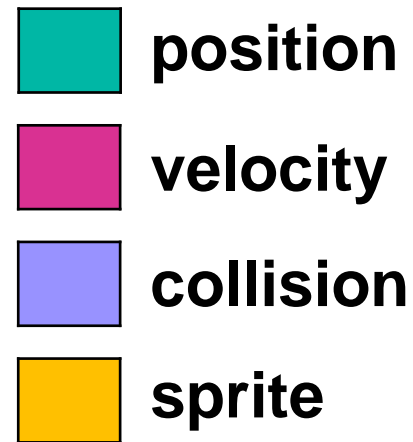
Memory & ECS

Where do we store our Components?

- Inside Entities?



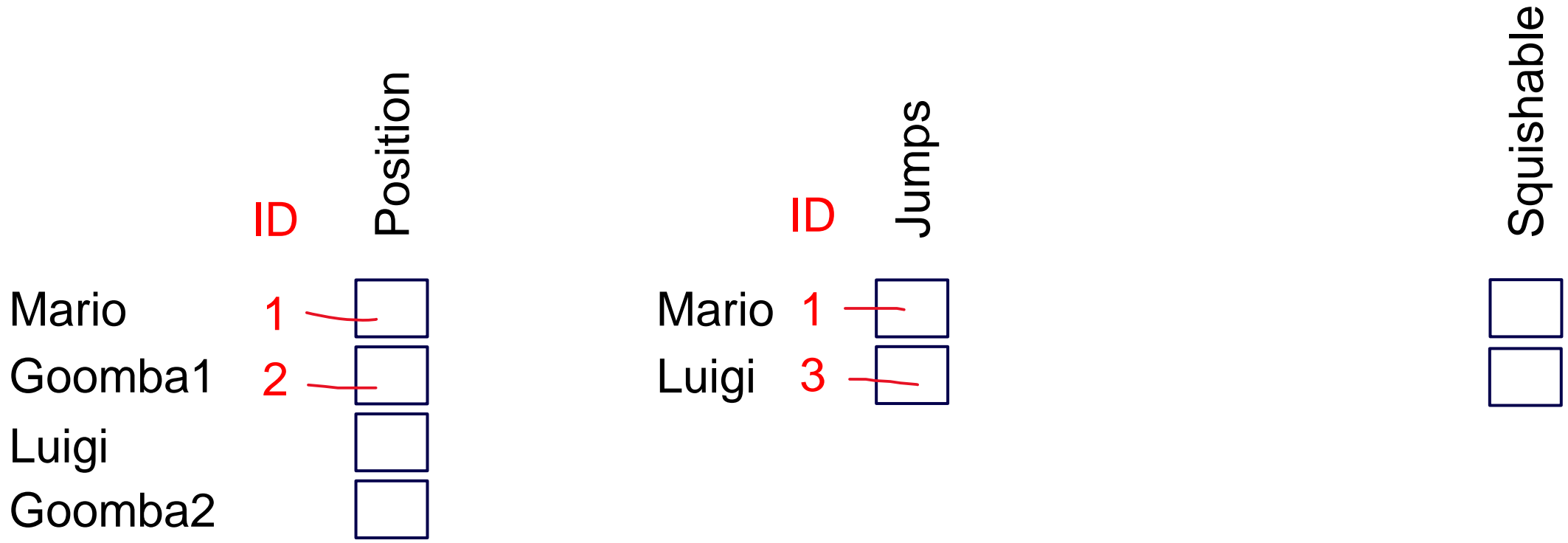
Memory Blocks



Update loop has to access non-contiguous memory repeatedly!

Slow memory access!

The Map Approach (entity ID to component address)

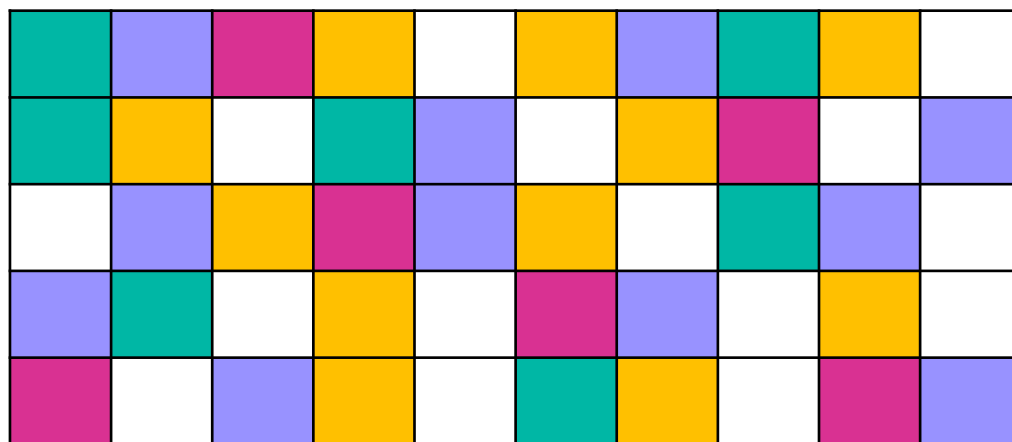


Concept: A (hierarchical) acceleration structure to lookup components
Implementation: `std::map<Entity,Position>`

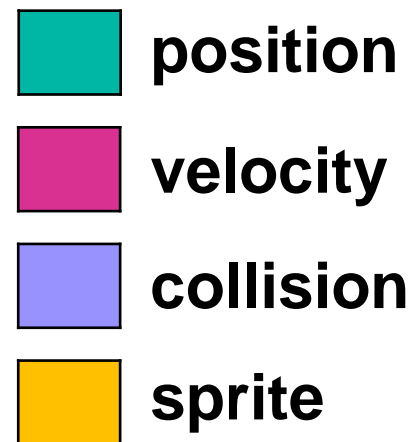
Memory & ECS

Where do we store our Components?

- In a map?



Memory Blocks



Update loop has to access non-contiguous memory repeatedly!

Slow memory access!

The (giant) Sparse Array

	ID	Position	Velocity	Jumps	Player	Squishable
Mario	1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Goomba1	2	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>
Luigi		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Goomba2		<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>

Issues?

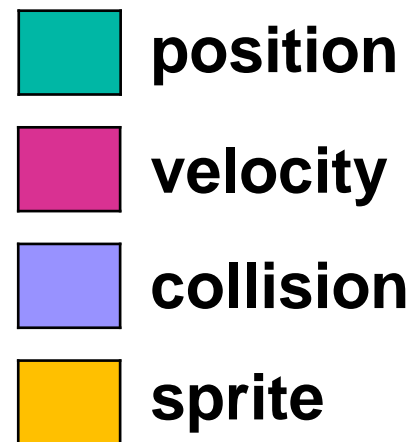
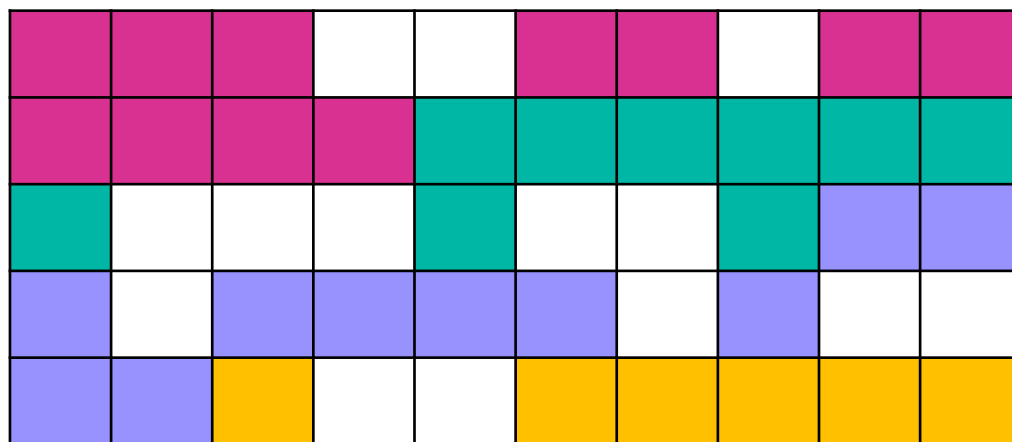
Concept: A huge data matrix of size Nr. Entities x Nr. components

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

Bitset / Bitmap

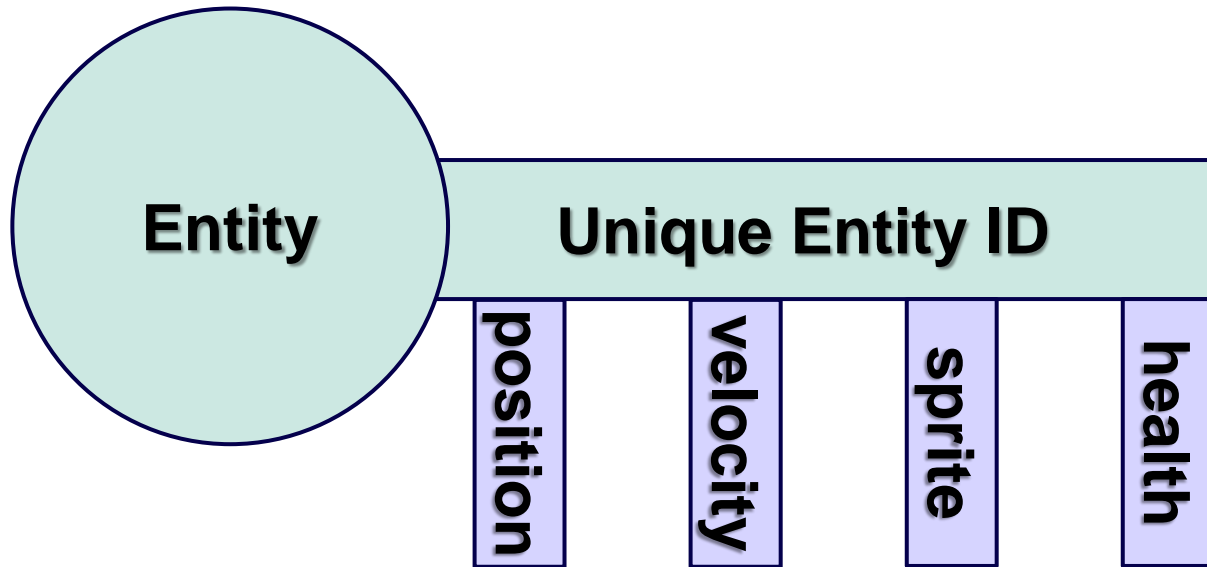
	ID	Bitset/bitmap	Position	Velocity	Jumps	Player	Squishable	Issues?
Mario	1	11110	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>		
Goomba1	2	11001	<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	
Luigi	3		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>			
Goomba2	4		<input type="checkbox"/>	<input type="checkbox"/>			<input type="checkbox"/>	

Concept: Each entity has a bitset that is true for its 'owned' components

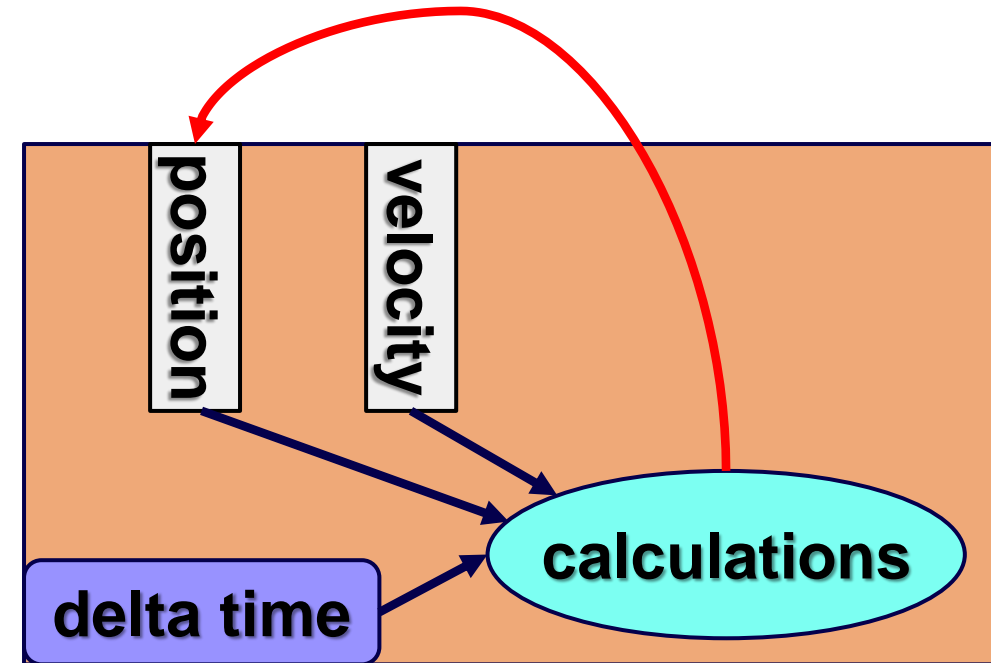
Implementation: long bitset; // how many components can we support?

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)

	ID	Position	Velocity	Jumps	Player	Squishable
Mario	1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Goomba1	2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Luigi		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Goomba2		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

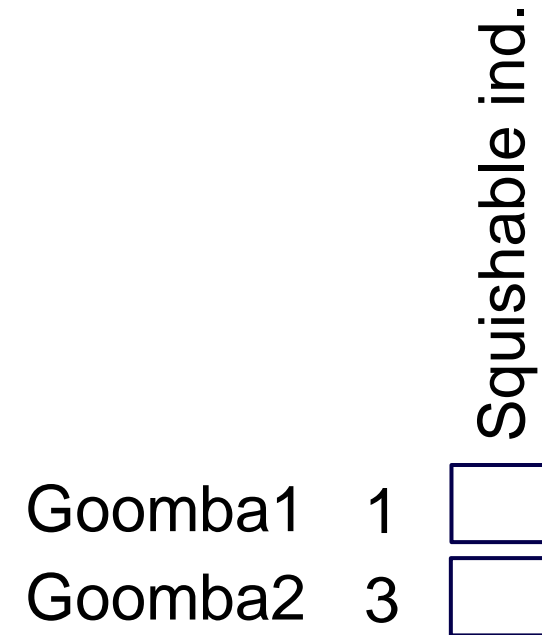
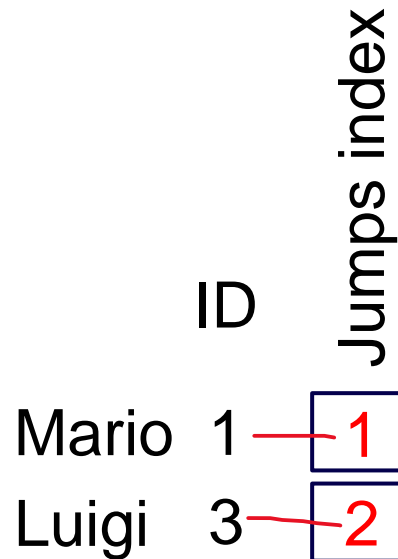
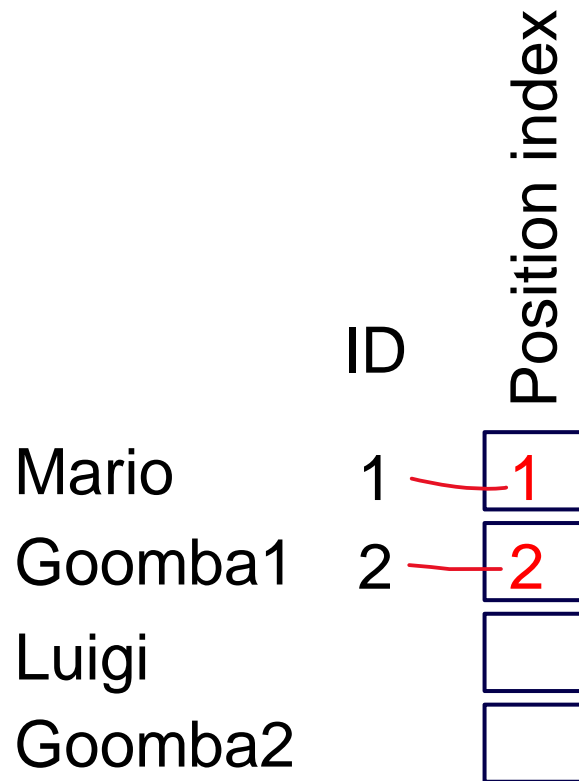
Issues?

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

Concept: One array/vector per component, **but how to associate?**

Implementation: `std::vector<Position>`; `std::vector<Velocity>` + X?

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

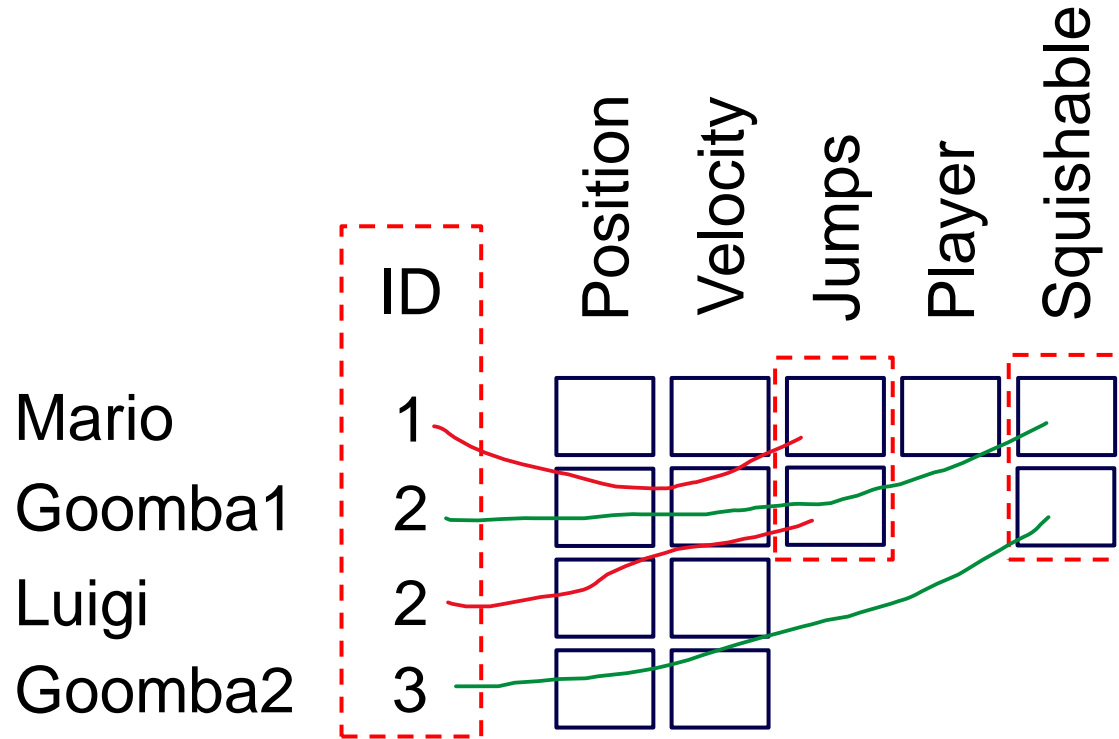


Issues?

Concept: Combine dense vectors with a map

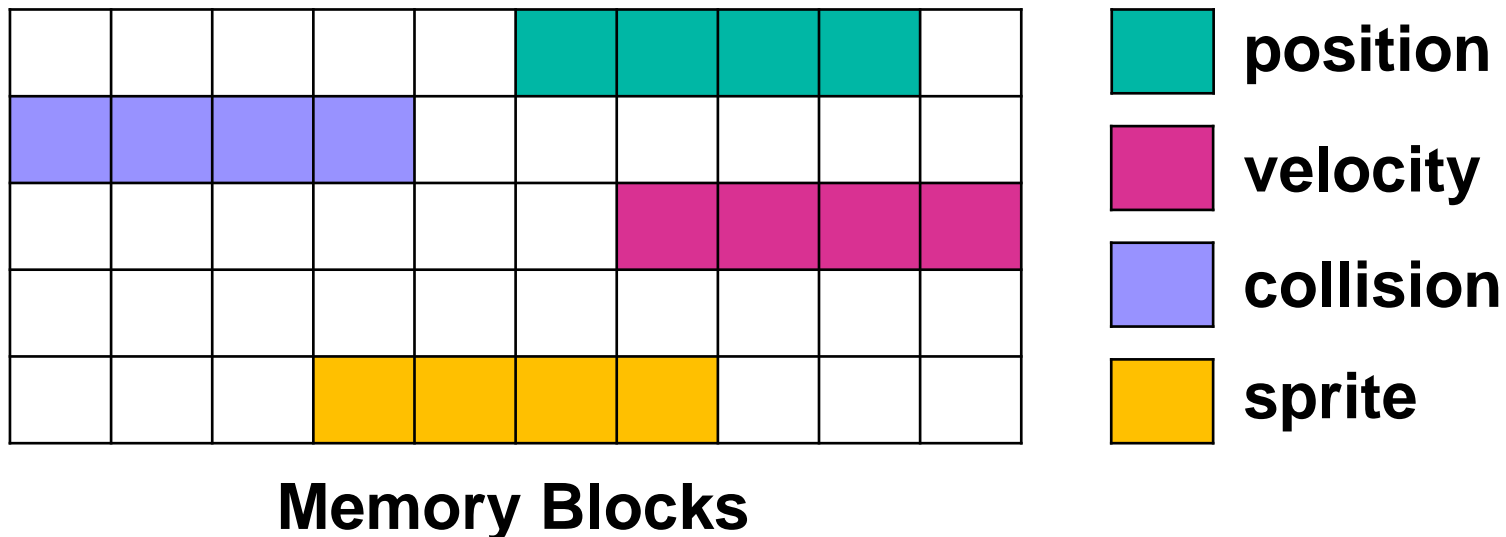
Implementation: `std::vector<Component>; std::map<Entity, unsigned int>`

Map + Dense Vector (different visualization)

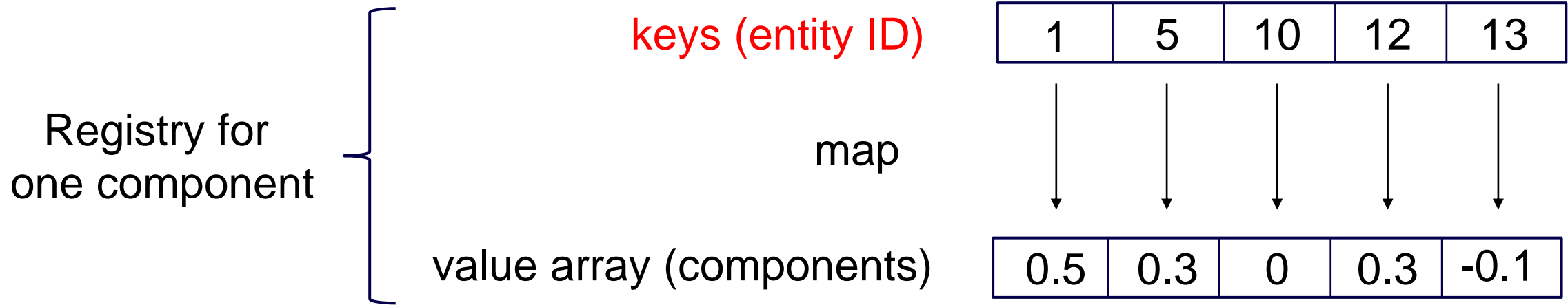


Cache is Key

- 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

```
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

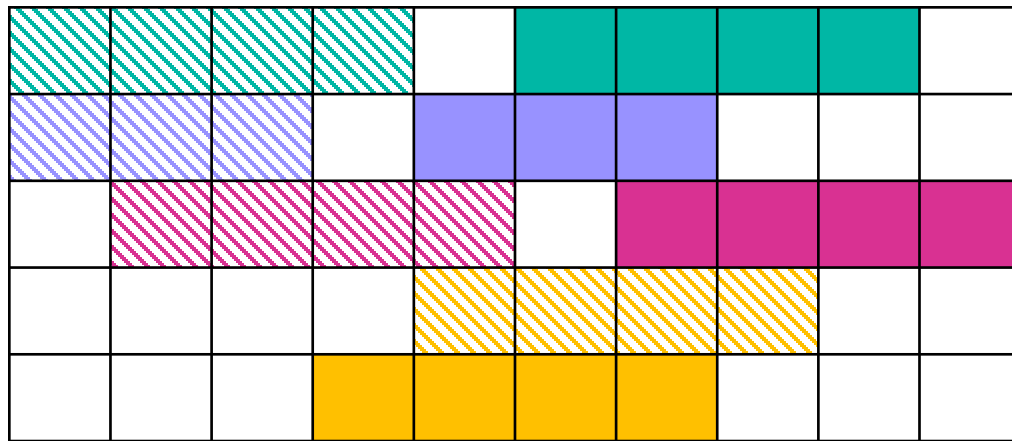
```
for(int entity : velocity_entities) // efficient
    if (position_entity_map.has(entity)) // inefficient lookup
        position_entity_map.get(entity) += velocity_entity_map.get(entity); // 2x inefficient lookup
```

We can access the velocity components in linear fashion





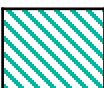

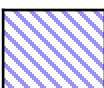

```
for(int vel_i = 0; vel_i < velocity_entities.size(); vel_i++) // efficient
    Entity entity : velocity_entities[vel_i]; // efficient
    int pos_i = position_entity_map.getIndex(entity); // inefficient lookup
    if (pos_i)
        position_components[pos_i] += reg_velocity_components[vel_i]; // efficient
```


Map + Component Vectors + Entity Vector

Cache is Key



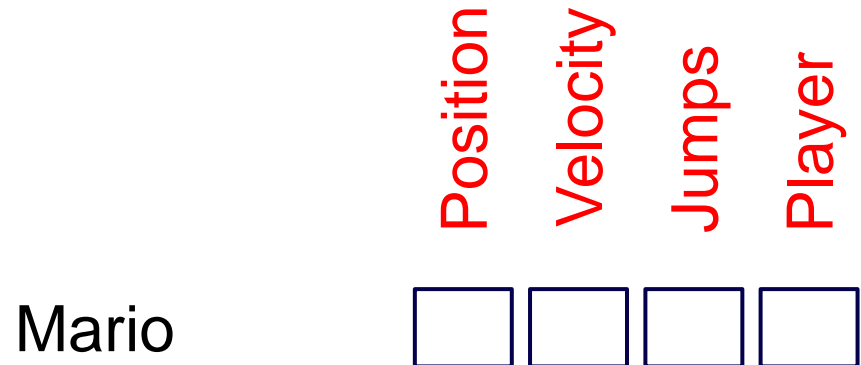
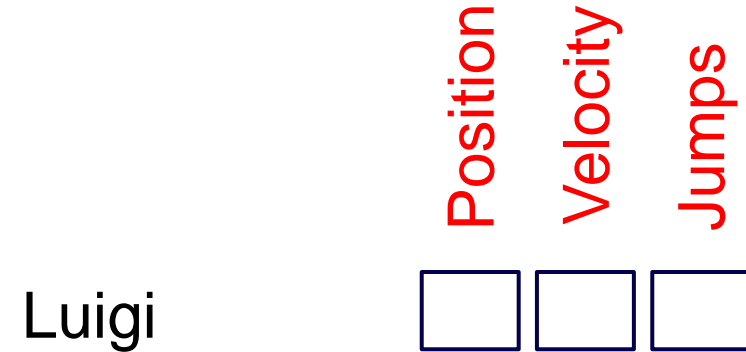
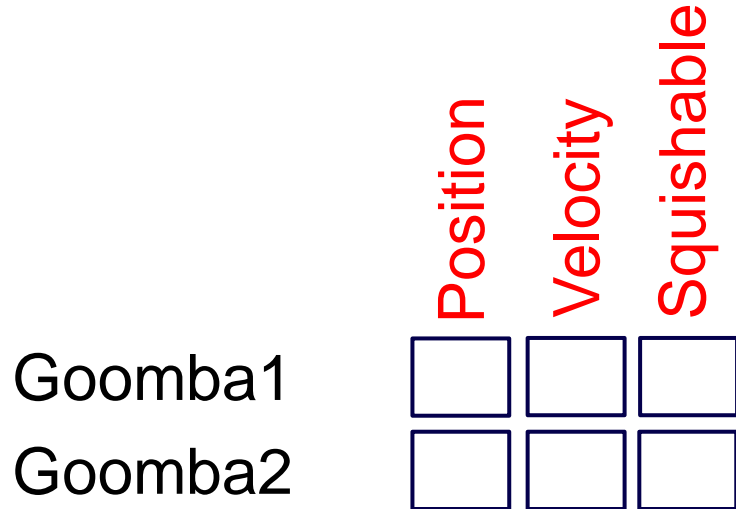
Memory Blocks

-  position
-  velocity
-  collision
-  sprite
-  position entities
-  velocity entities
-  collision entities
-  sprite entities

Update loop
accesses contiguous
memory **IDEAL!**

Map access
slow

Advanced ECS: Archetypes / prototypes / pools



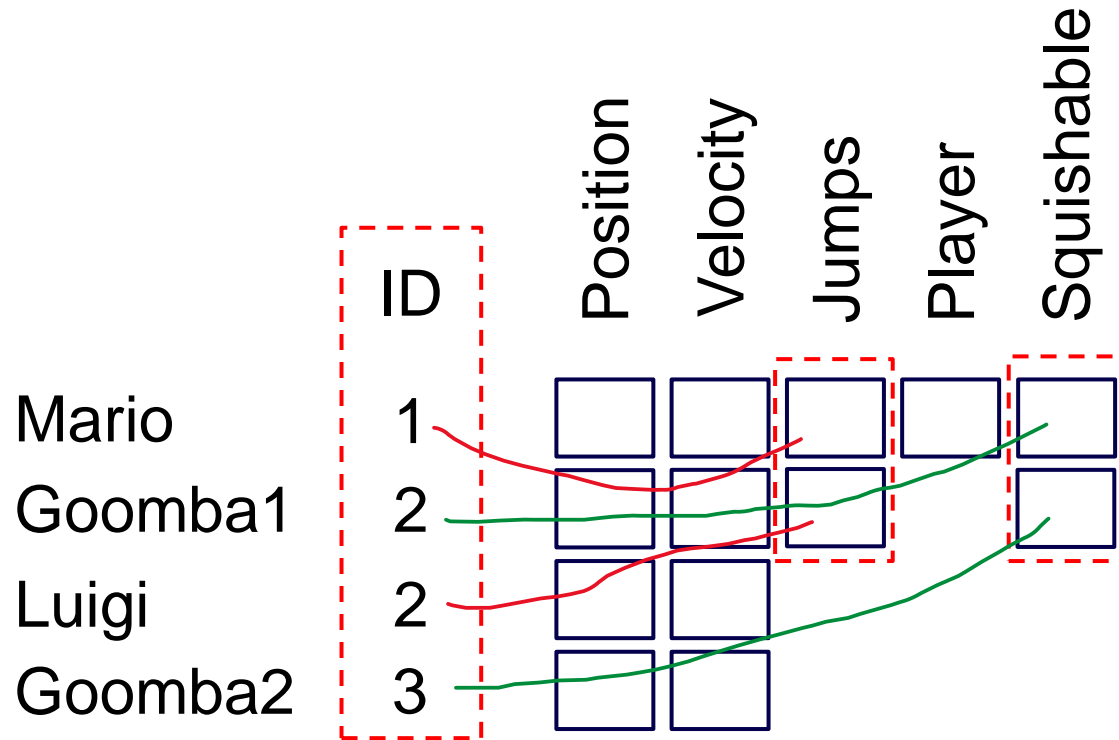
- **Concept:** store all types with the same components in dense arrays
- Used by the Unity ECS system
- Difficult to implement

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'

	ID	Index Pos	Index Vel	Index Jump	Index Player	Index Squish	Position	Velocity	Jumps	Player	Squishable
Mario	1			1	1						
Goomba1	2					1					
Luigi	3			2							
Goomba2	4					2					

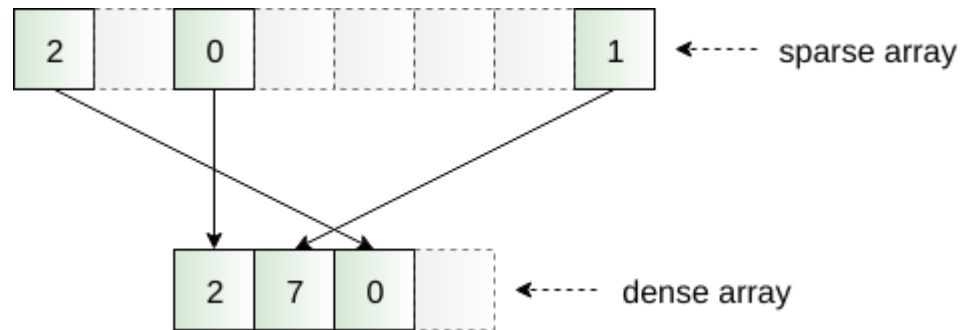
Issues?

Concept: Sparse array + dense array

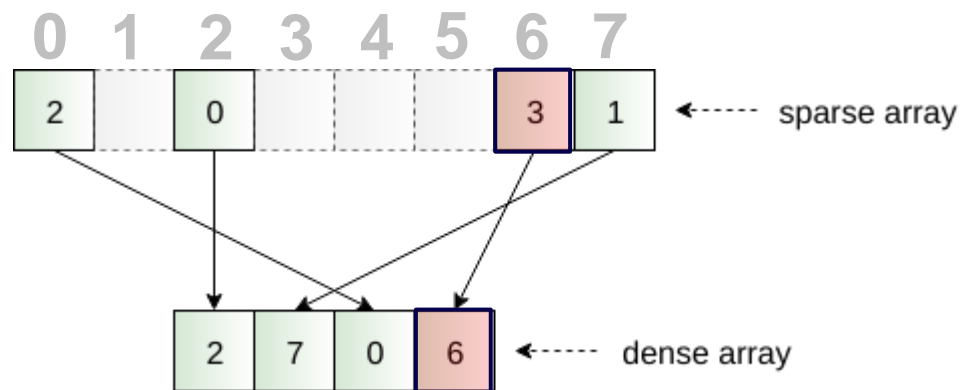
Implementation: `std::vector<Entity> entities; std::vector<unsigned int> indices; std::vector<Components> components;`

Self-study: Faster Lookup with Sparse Sets

Lookup:



Insert:



The map lookup (`map.get(entity)`) is costly

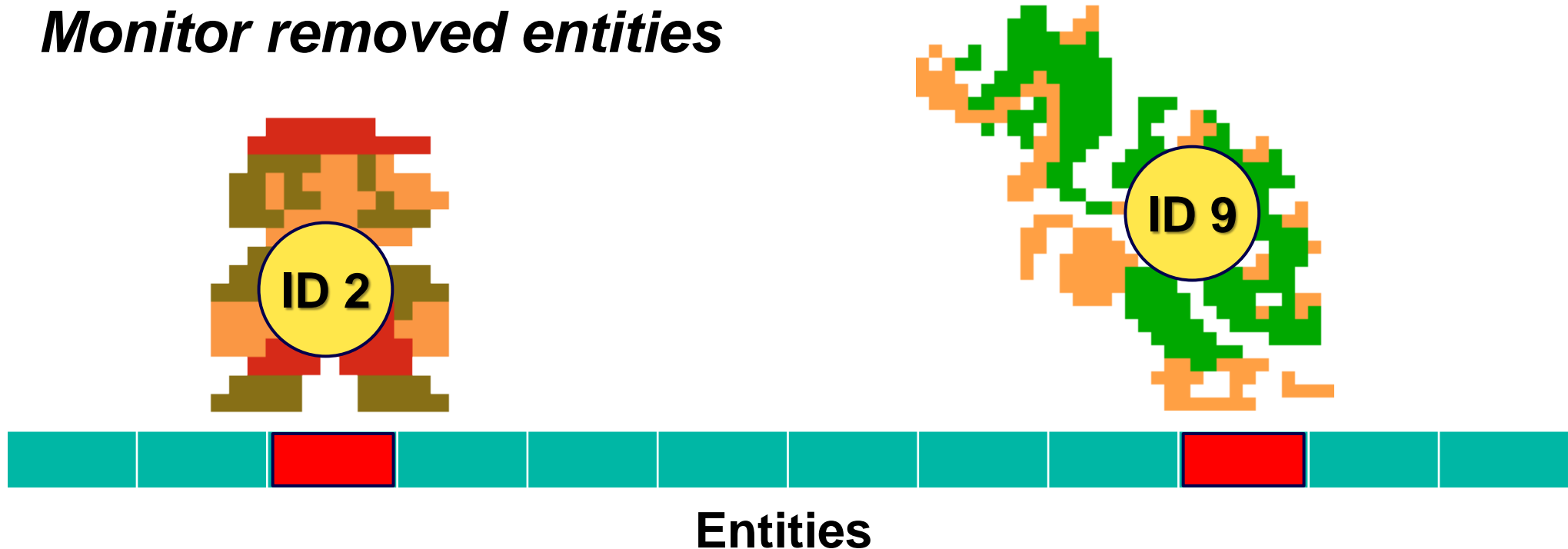
- A hashmap is $O(1)$, but that 1 is big

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*

← *Also simulation forms!*

2. *Environment reaction*

3. *Equations of Motion*

- sum forces & torques, solve for accelerations: $\vec{F} = ma$

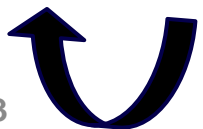
4. *Numerical integration*

- update positions, velocities

*We will have a separate
lecture on physics
simulation!*

5. *Collision detection*

6. *Collision resolution*



Our game loop (A1, main.cpp)

```
// 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
