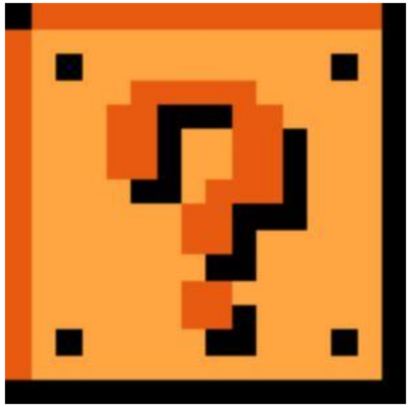
# CPSC 427
# Video Game Programming

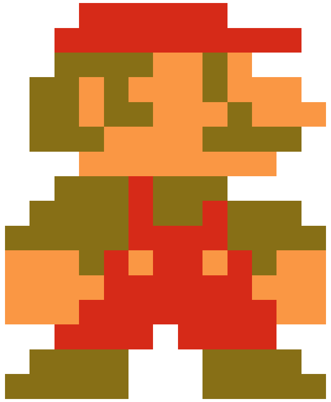
## Entity Component System (ECS)

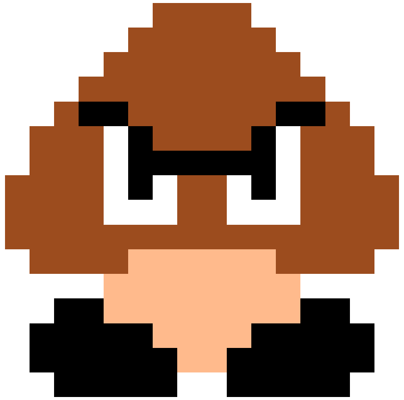
ECS is used in Minecraft and many other commercial games

# What are Entities?

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



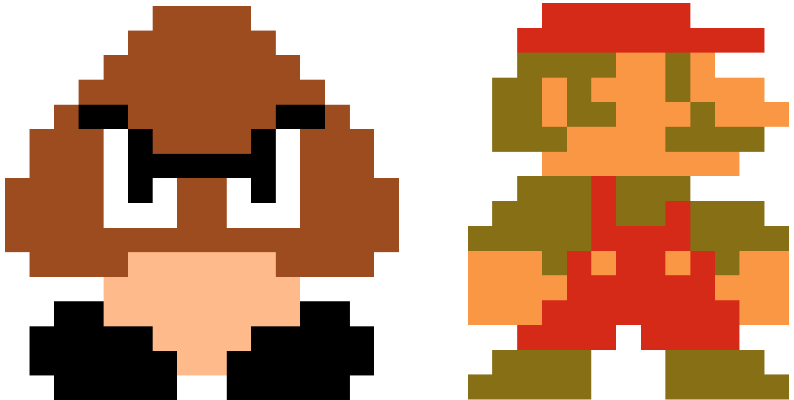


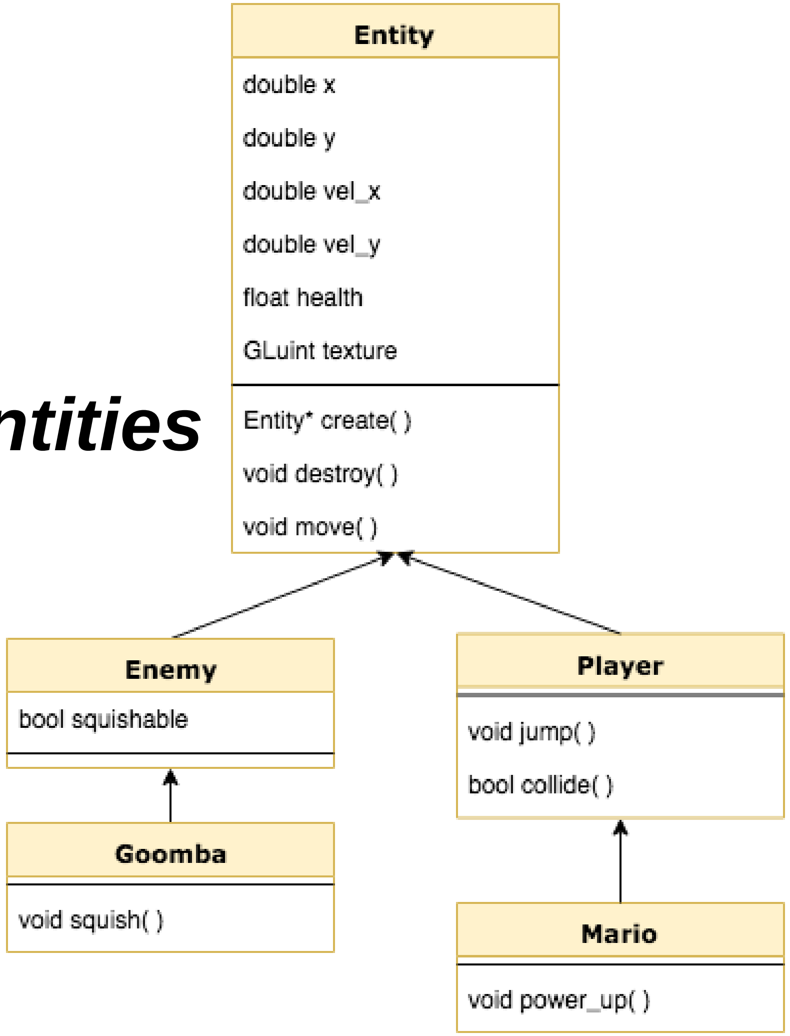


BGM

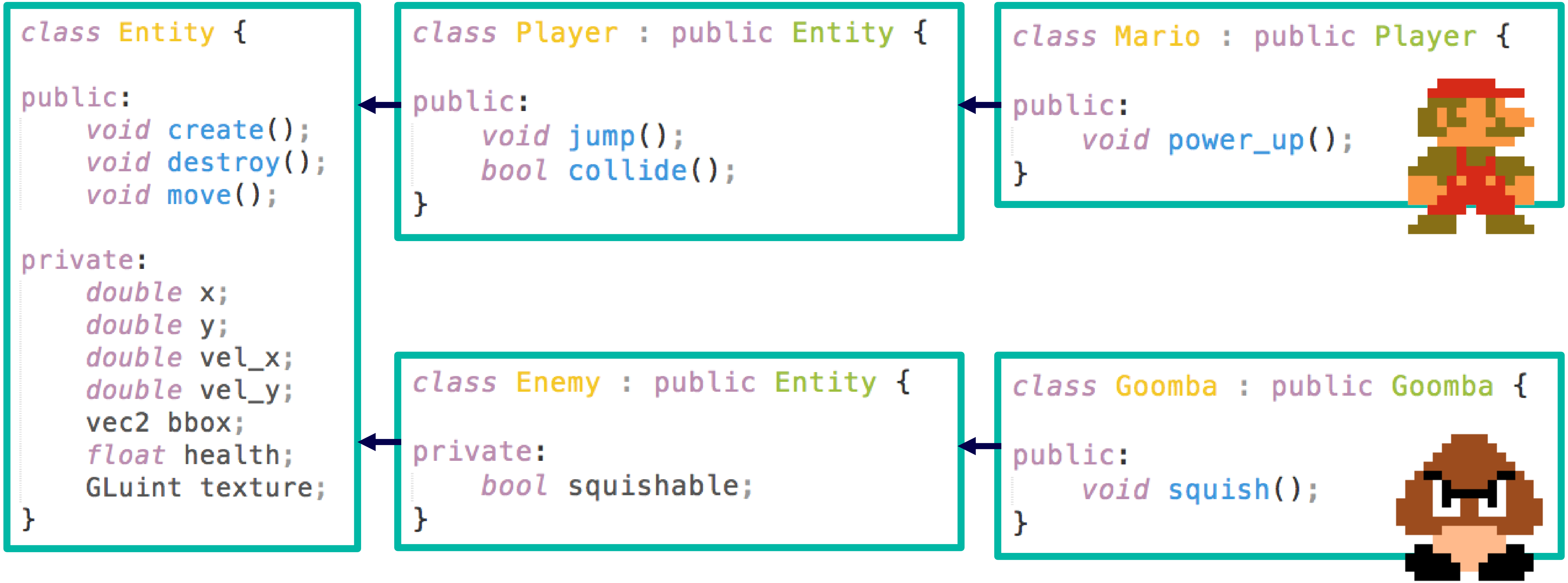
# Entities in Traditional Game Programming

- Object-Oriented Programming
  - *Entities as objects*
    - Contains data, behaviors, etc.
  - *Entity Hierarchy: Entities extend other Entities*





# Entity Hierarchy (object oriented design)



```cpp
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;
}
```

```cpp
class Player : public Entity {

public:
    void jump();
    bool collide();
}
```

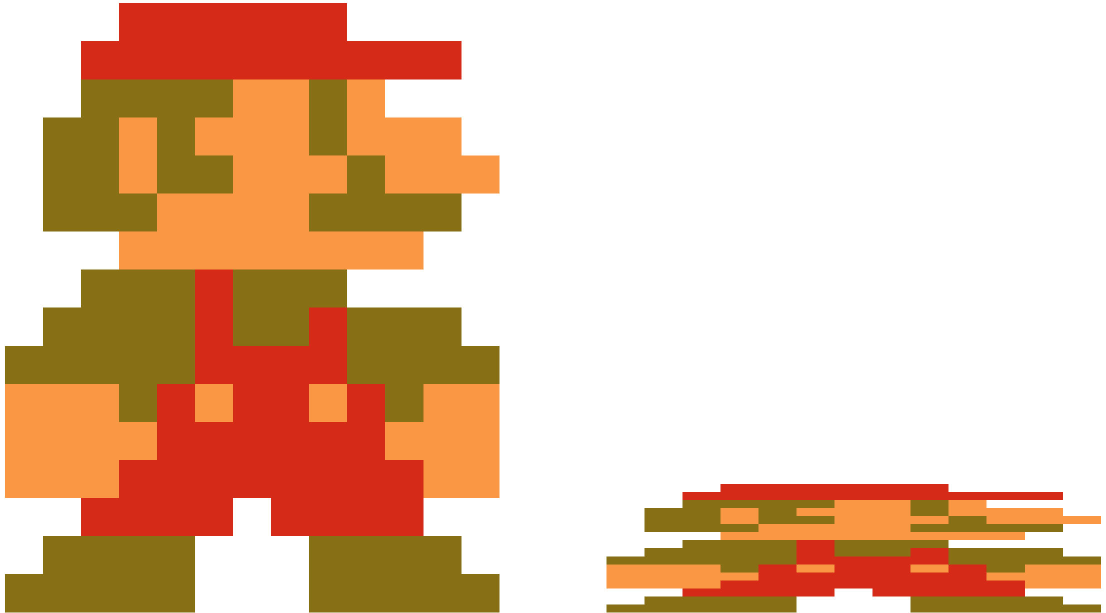
```cpp
class Mario : public Player {

public:
    void power_up();
}
```

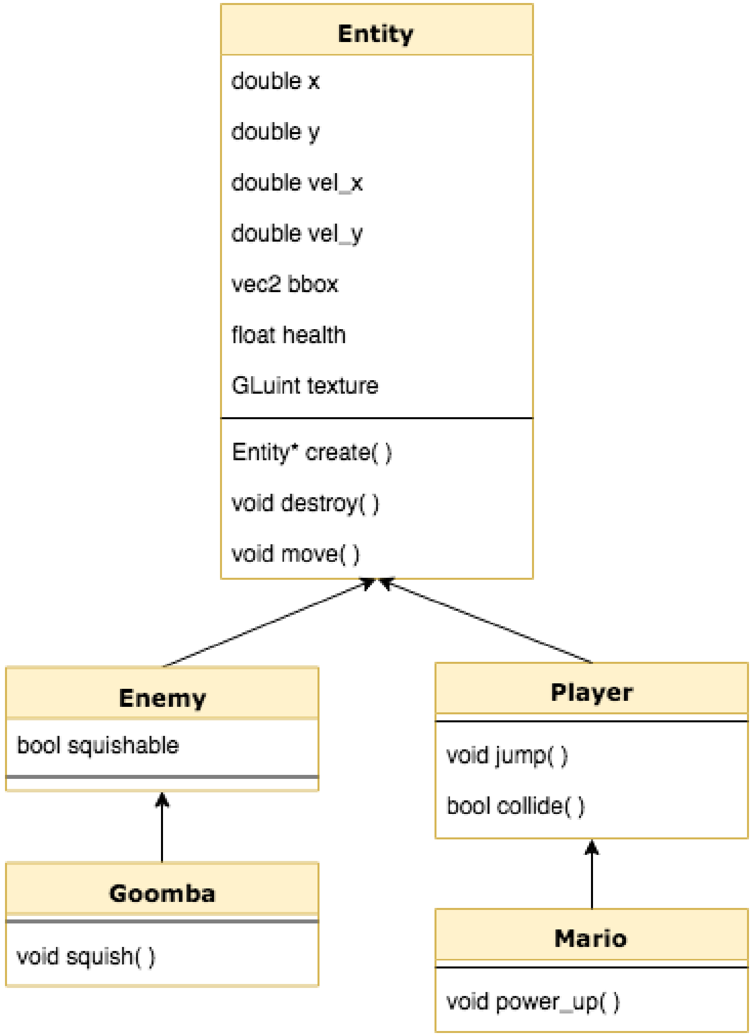
```cpp
class Enemy : public Entity {

private:
    bool squishable;
}
```

```cpp
class Goomba : public Goomba {

public:
    void squish();
}
```

# Issues with Object-Oriented Approach

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





**Entity**
double x
double y
double vel_x
double vel_y
vec2 bbox
float health
GLuint texture

Entity* create( )
void destroy( )
void move( )

**Enemy**
bool squishable

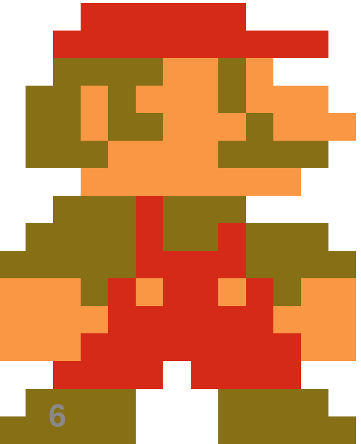
**Player**
void jump( )
bool collide( )

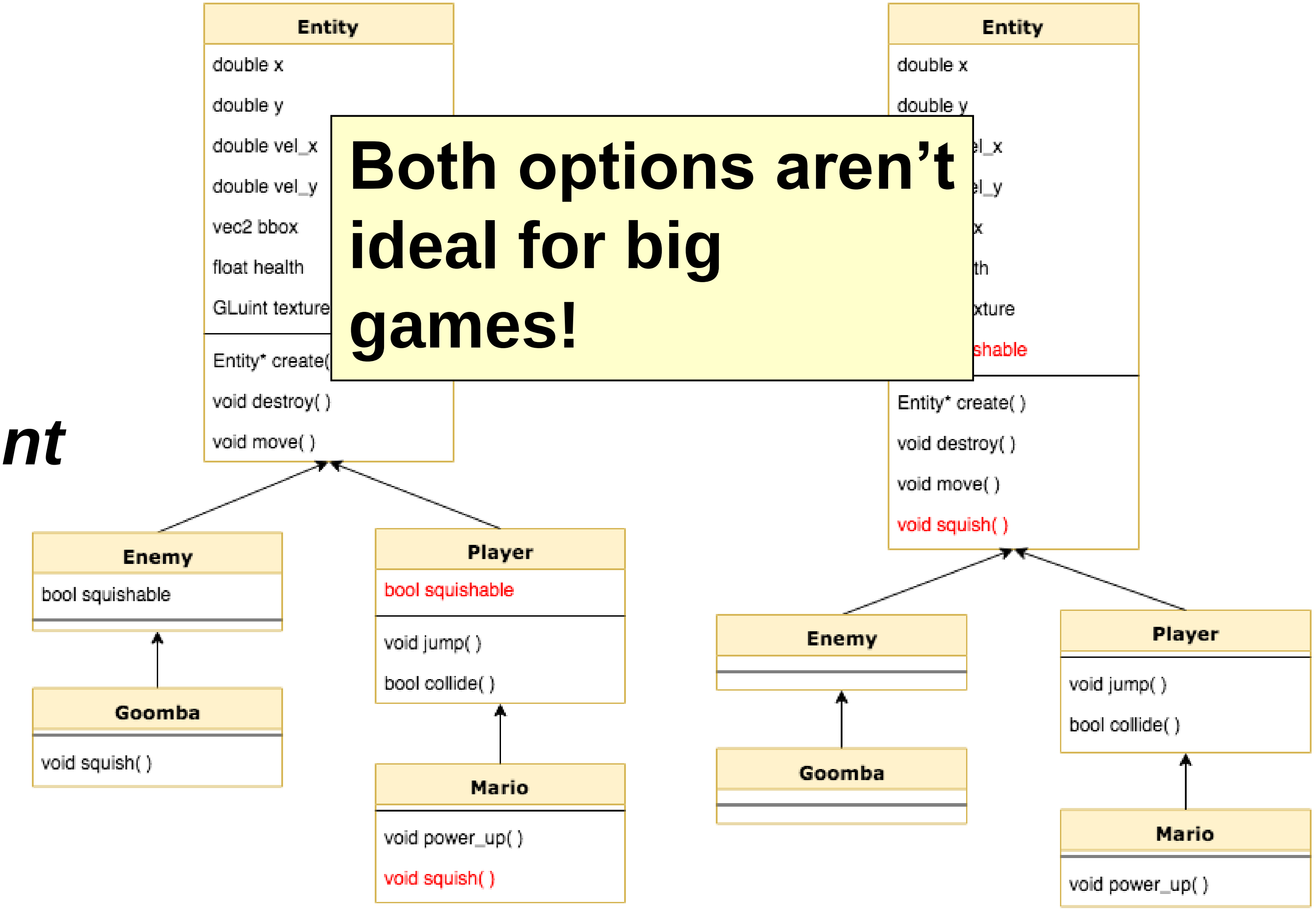
**Goomba**
void squish( )

**Mario**
void power_up( )

# Issues with Object-Oriented Approach

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





Entity

double x

double y

double vel_x

double vel_y

vec2 bbox

float health

GLuint texture

Entity* create(

void destroy( )

void move( )

Enemy

bool squishable

Goomba

void squish( )

Player

bool squishable

void jump( )

bool collide( )

Mario

void power_up( )

void squish( )

Entity

double x

double y

el_x

el_y

x

th

xture

shable

Entity* create( )

void destroy( )

void move( )

void squish( )

Enemy

Goomba

Player

void jump( )

bool collide( )

Mario

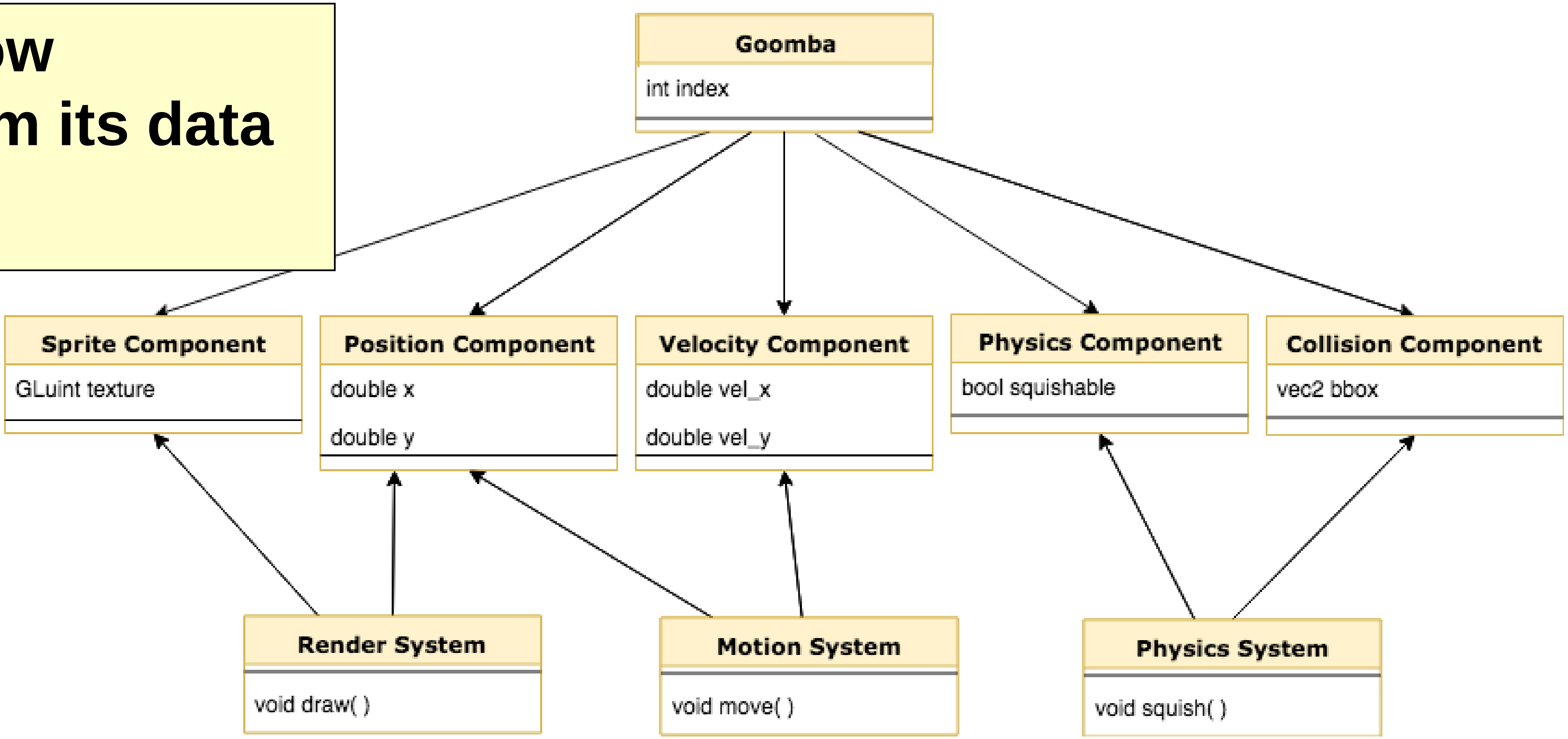
void power_up( )

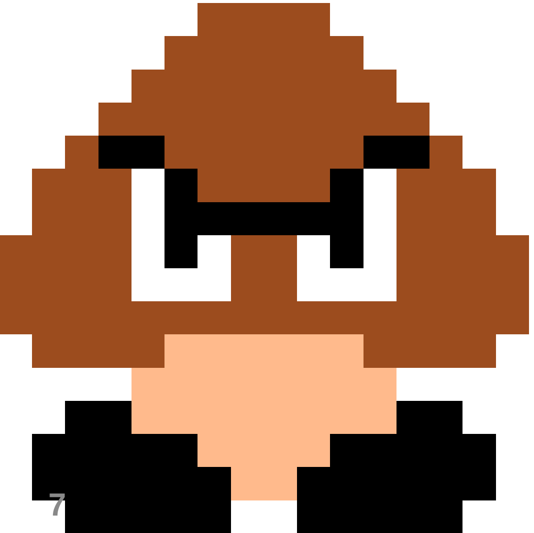
**Both options aren’t ideal for big games!**

© Alla Sheffer, Helge Rhodin

# Example ECS Diagram

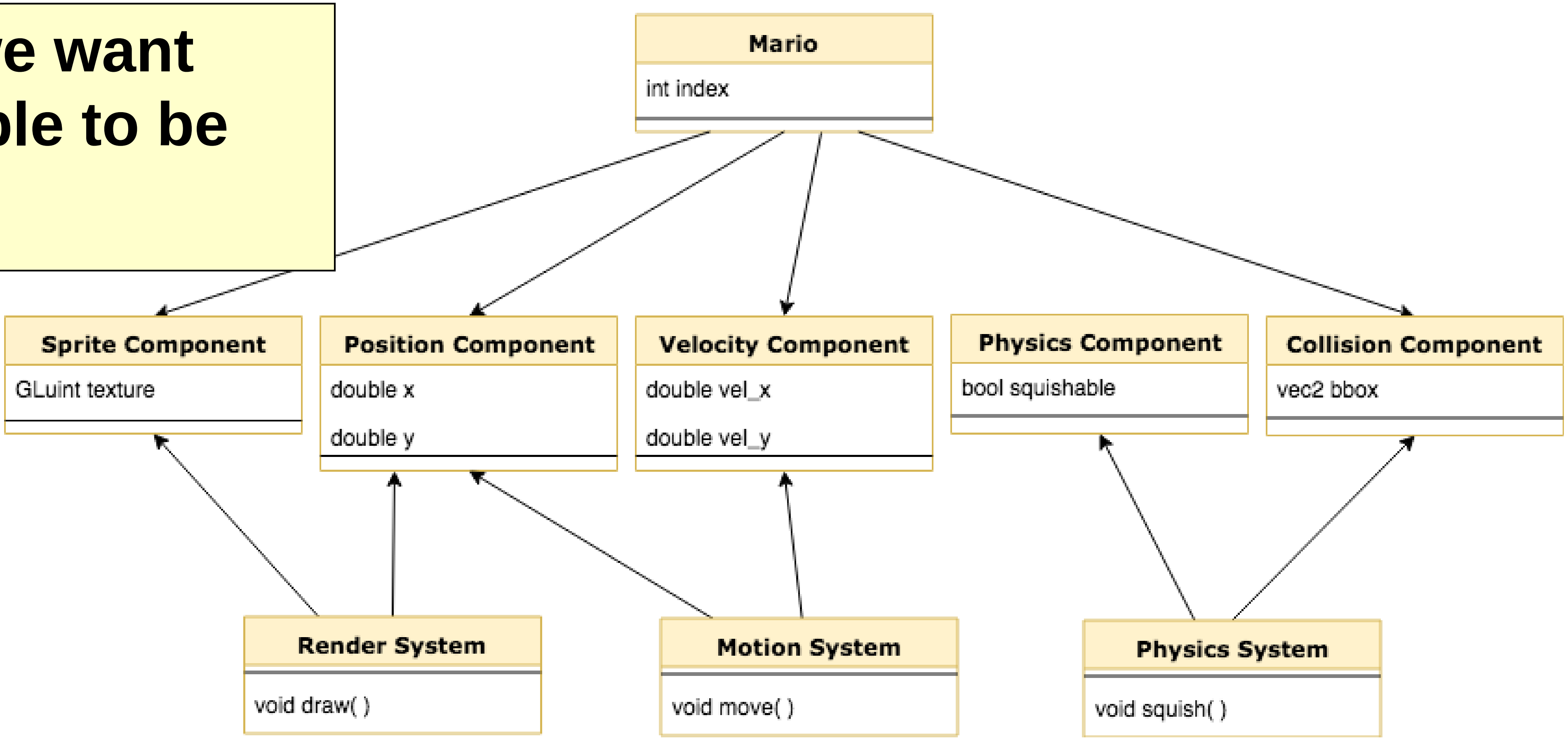
Goomba is now separated from its data & methods





# Example ECS Diagram

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



Mario
int index

Sprite Component
GLuint texture

Position Component
double x
double y

Velocity Component
double vel_x
double vel_y

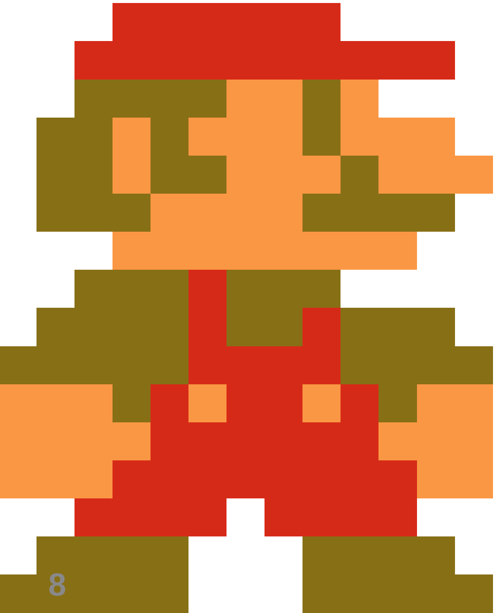
Physics Component
bool squishable

Collision Component
vec2 bbox

Render System
void draw( )

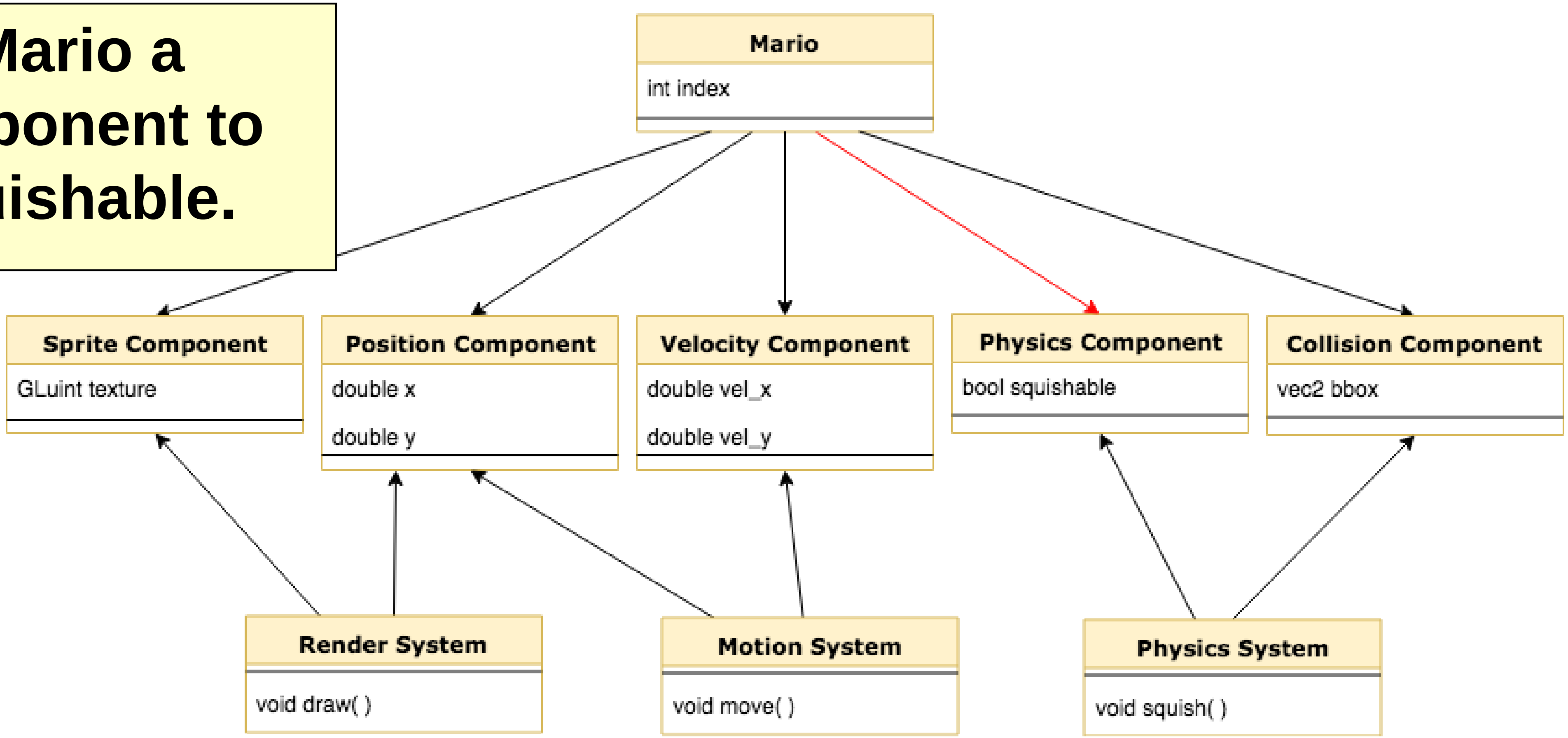
Motion System
void move( )

Physics System
void squish( )



# Example ECS Diagram

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



Mario
int index

Sprite Component
GLuint texture

Position Component
double x
double y

Velocity Component
double vel_x
double vel_y

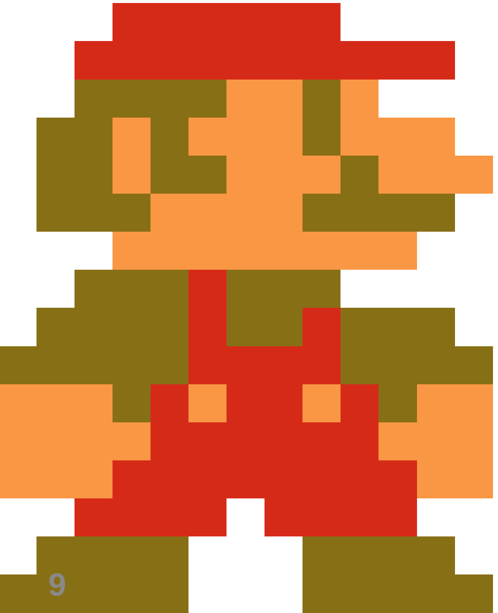
Physics Component
bool squishable

Collision Component
vec2 bbox

Render System
void draw( )

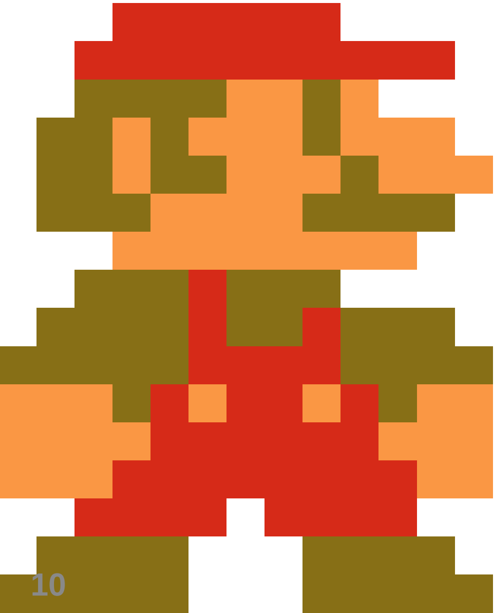
Motion System
void move( )

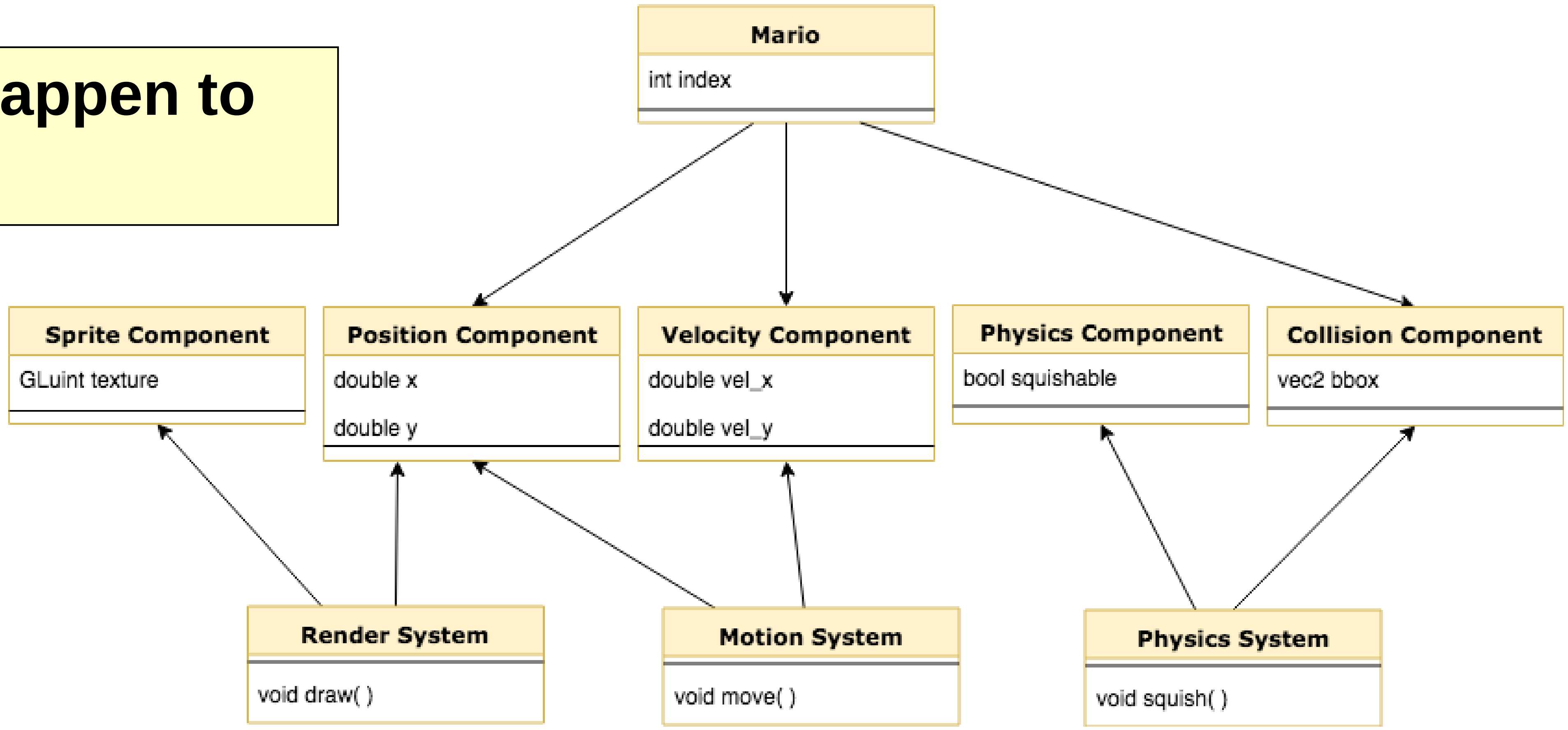
Physics System
void squish( )



# Example ECS Diagram

**What would happen to Mario here?**





**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

**Collision Component**
vec2 bbox

**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

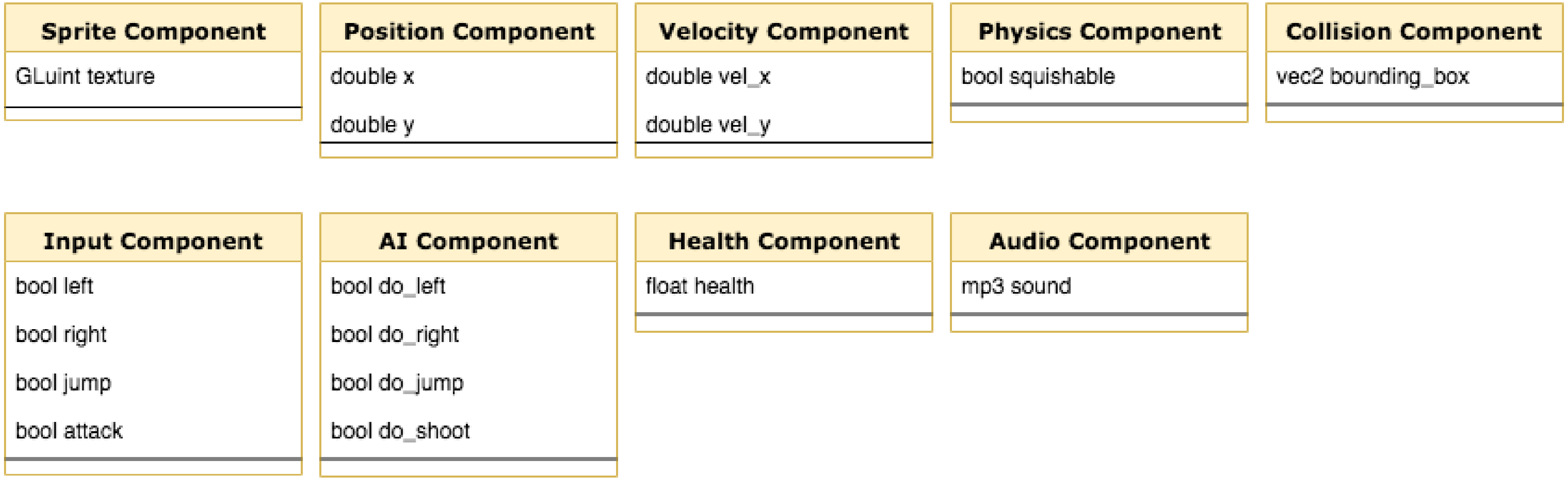
# What is ECS?

- **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*



**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

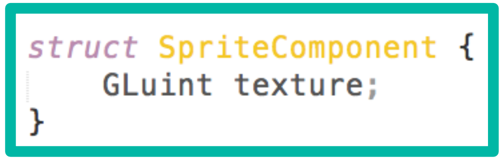
**AI Component**
bool do_left
bool do_right
bool do_jump
bool do_shoot

**Health Component**
float health

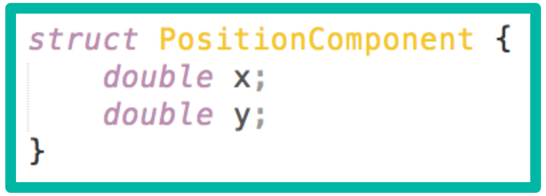
**Audio Component**
mp3 sound

# Component

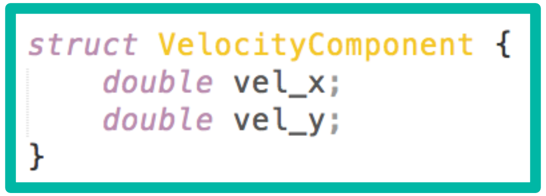
- **Typically implemented with structs.**



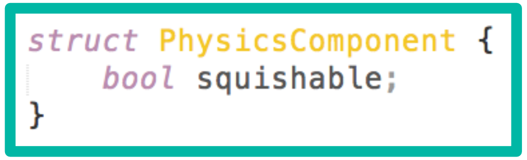
```
struct SpriteComponent {
    GLuint texture;
}
```



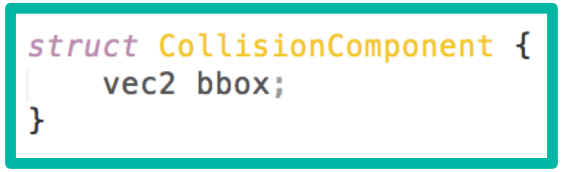
```
struct PositionComponent {
    double x;
    double y;
}
```



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



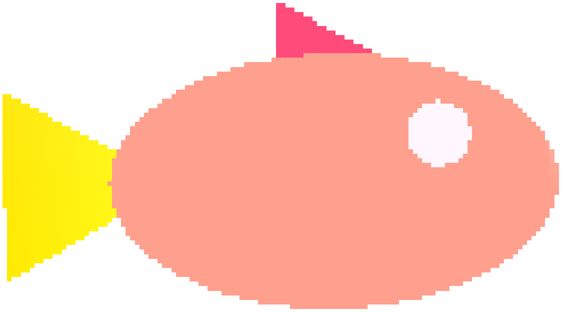
```
struct PhysicsComponent {
    bool squishable;
}
```



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

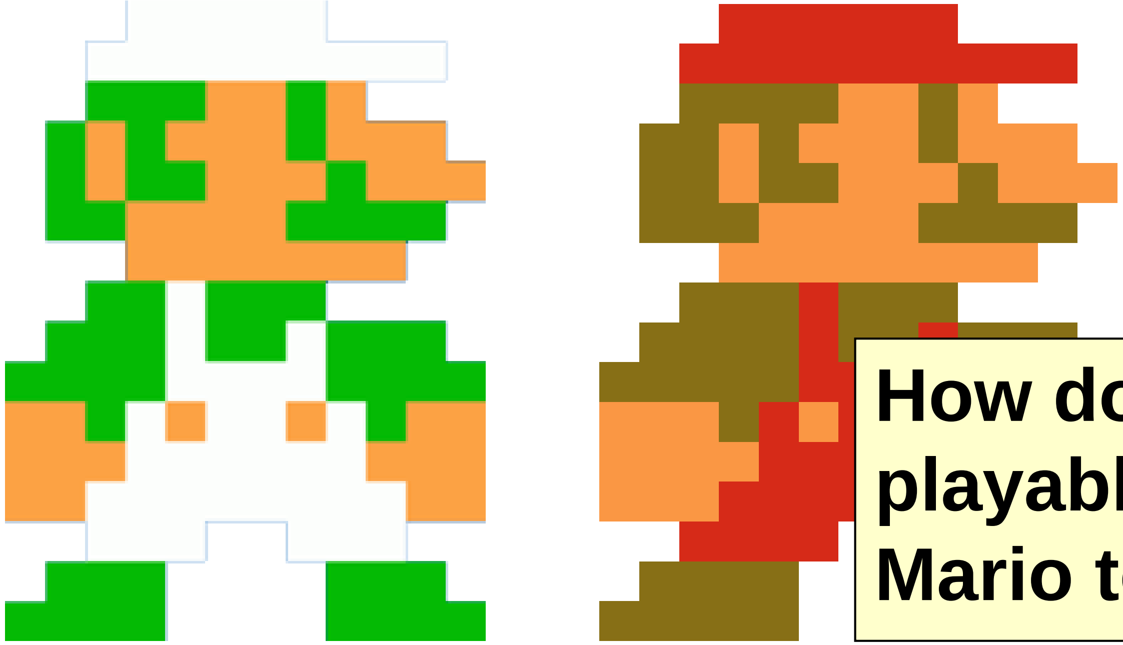
# What Components to Make?

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



# Components

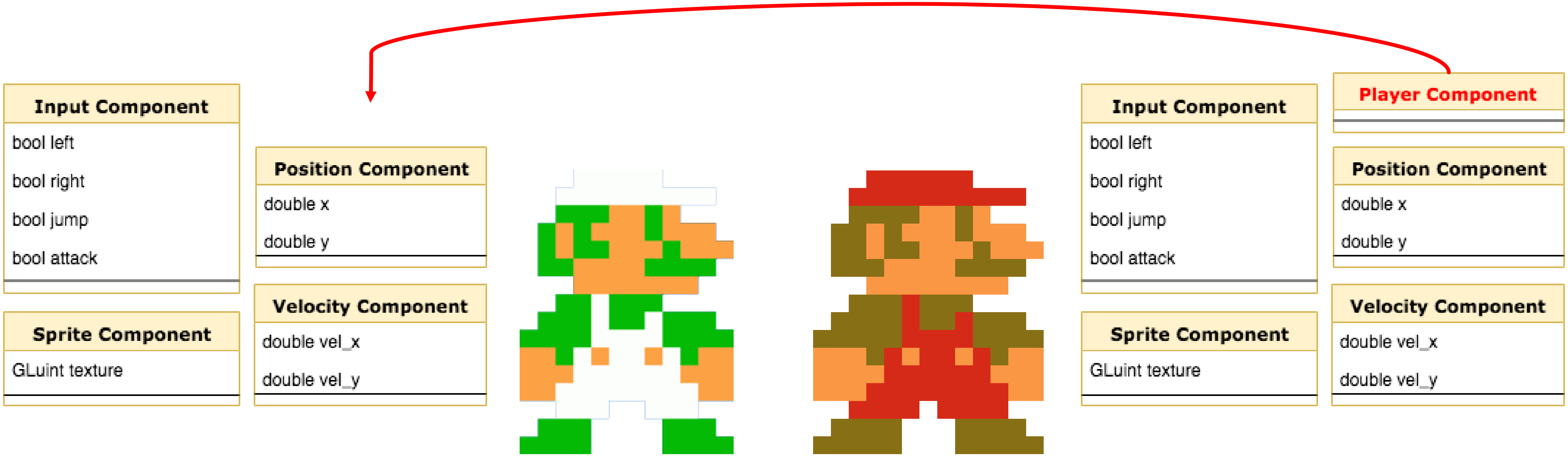
- **Easy to add new Entity characteristics**
  - ***Just create the desired Component & give to Entity***



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

# Components

- **Empty Components can be used to tag Entities**



**Input Component**

bool left

bool right

bool jump

bool attack

**Sprite Component**

GLuint texture

**Position Component**

double x

double y

**Velocity Component**

double vel_x

double vel_y

**Input Component**

bool left

bool right

bool jump

bool attack

**Sprite Component**

GLuint texture

**Player Component**

**Position Component**

double x

double y

**Velocity Component**

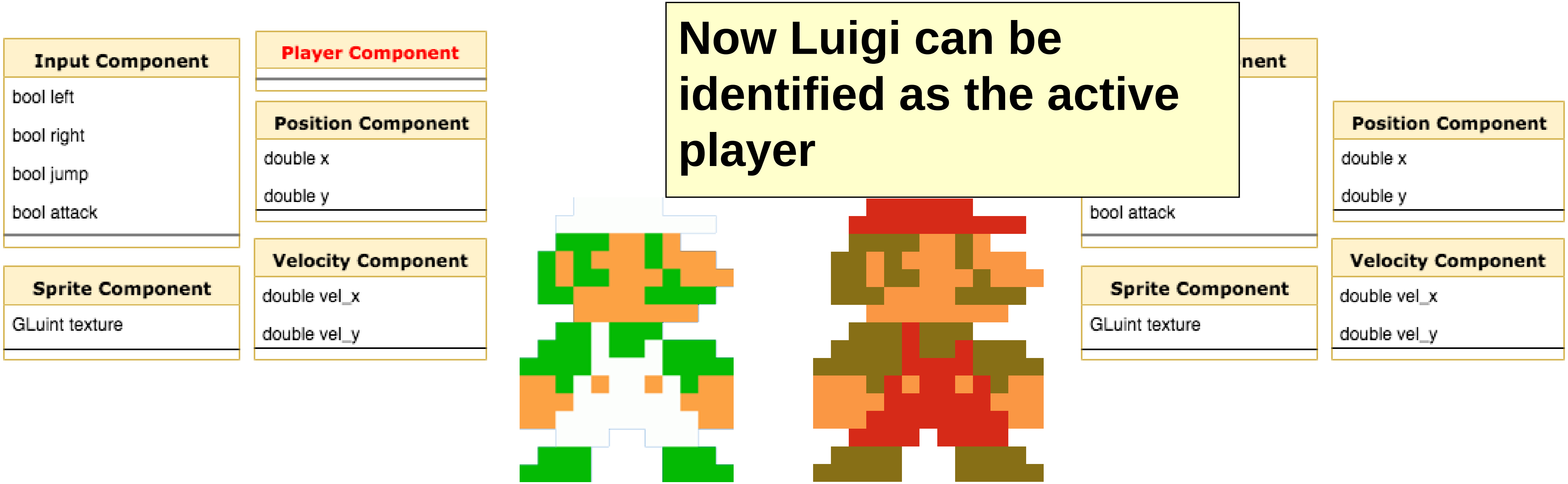
double vel_x

double vel_y

**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**

**nent**

bool attack

**Sprite Component**

GLuint texture

**Position Component**

double x

double y

**Velocity Component**

double vel_x

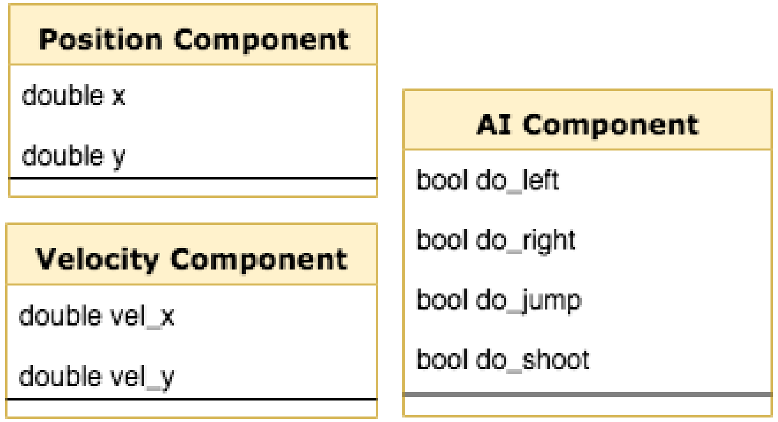
double vel_y

# 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

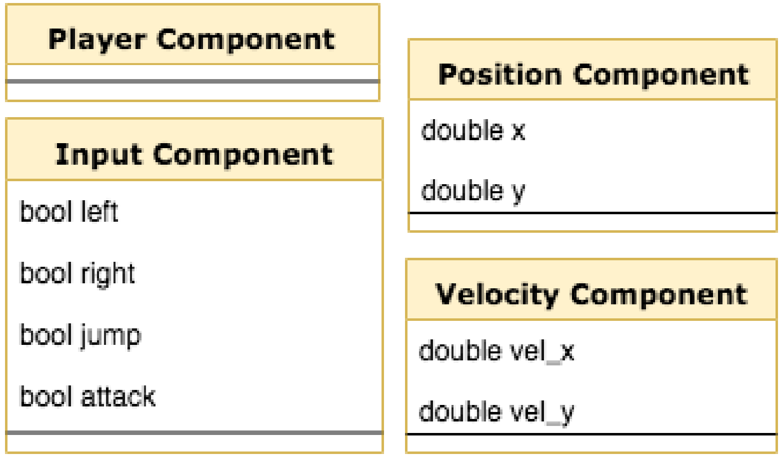
- **What systems might these related groups of components describe?**



**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**

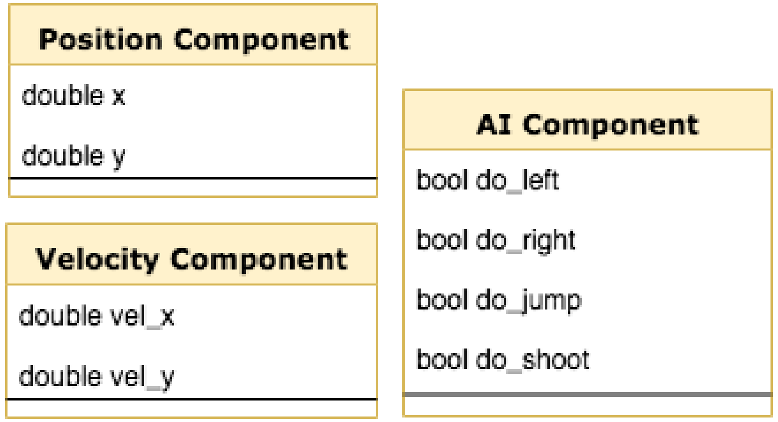
**Input Component**
- bool left
- bool right
- bool jump
- bool attack

**Position Component**
- double x
- double y

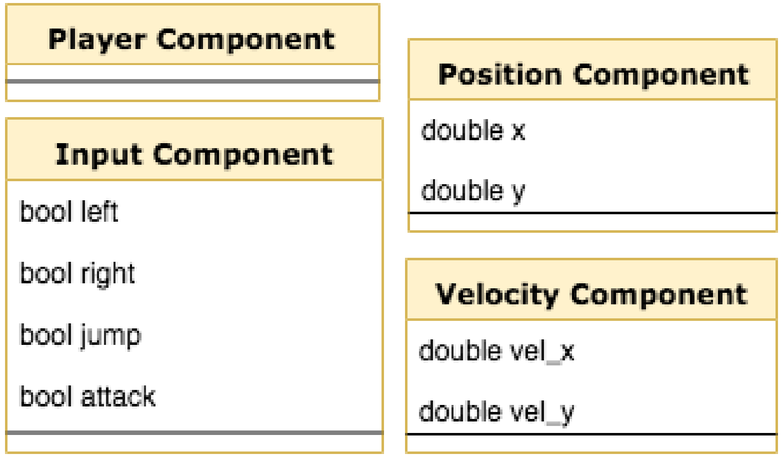
**Velocity Component**
- double vel_x
- double vel_y

# 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 : registry<Velocity>.components)
    velocity += 9.81 * dt
```

*The physics system does not care about entities at all!*

## Game loop

```
Entity player;
    if(! player.has<Alive>() ) exit();
```

*Single boolean check*

## Motion System … iterates over all entities that have velocity and position

```
for(Entity entity : registry< Velocity >.entities)
    if(entity.has< Position>() )
            entity.get<Position>() += entity.get<Velocity>()
```

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

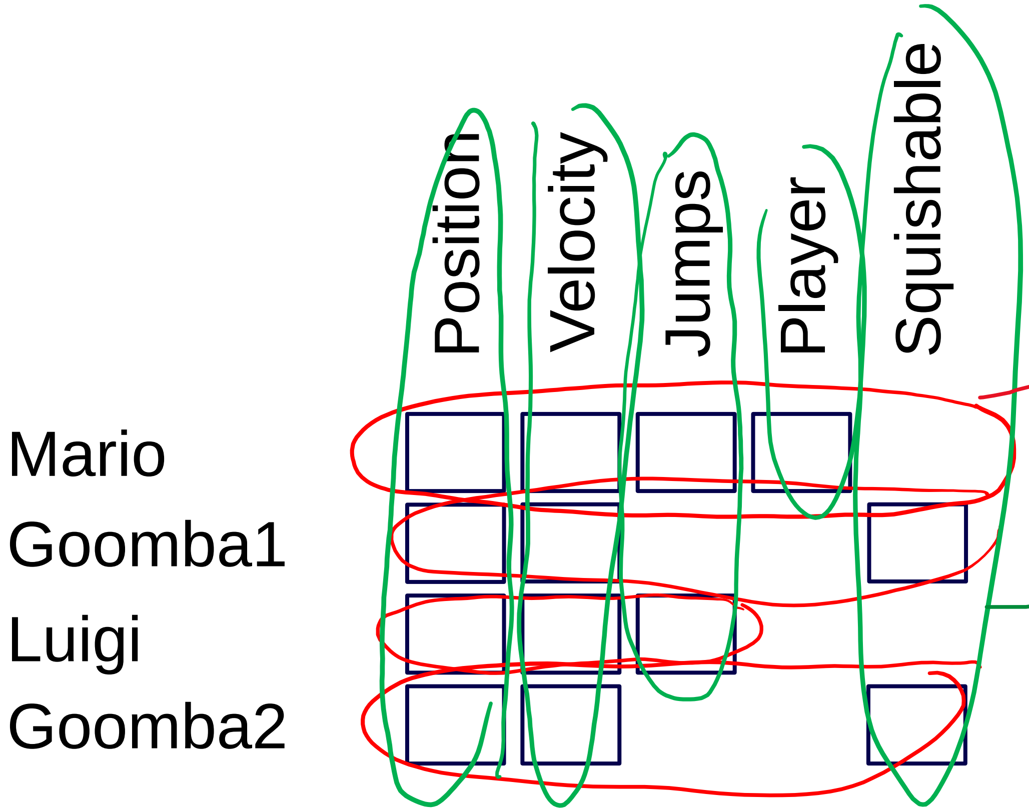
# ECS implementations

# Memory & ECS

**Where do we store our Components?**

- **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

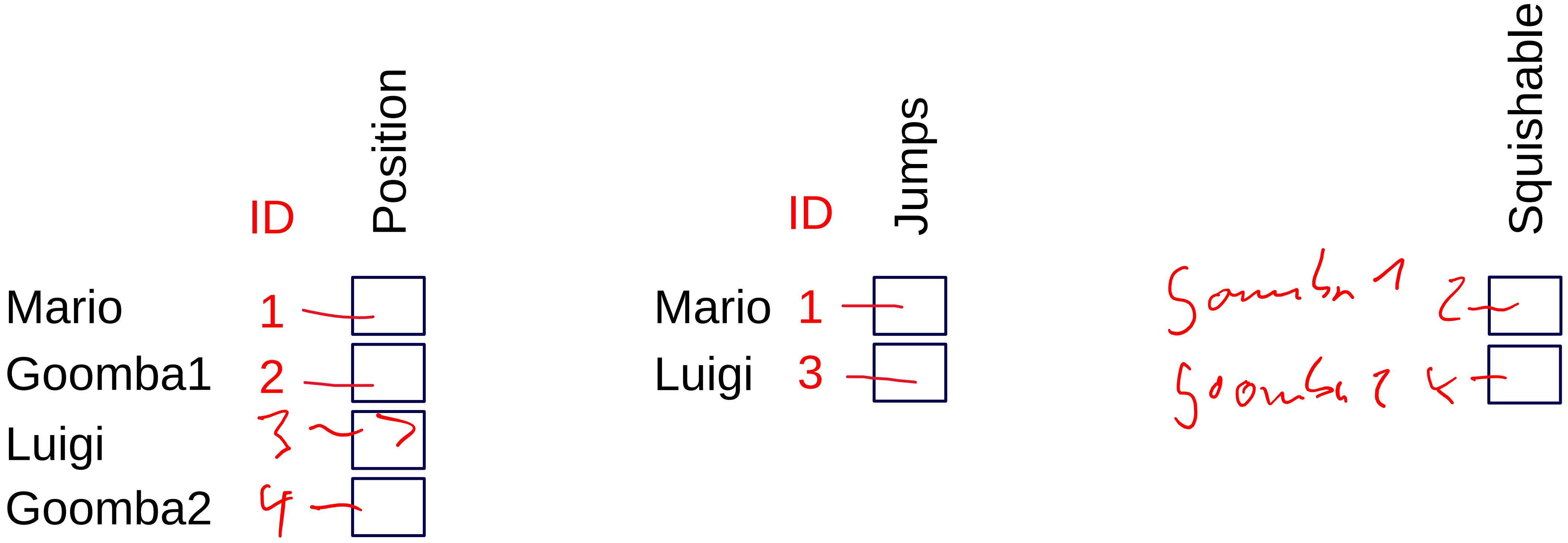


|  | Position | Velocity | Jumps | Player | Squishable |
|---|---|---|---|---|---|
| Mario | □ | □ | □ | □ |  |
| Goomba1 | □ | □ |  |  | □ |
| Luigi | □ | □ | □ |  |  |
| Goomba2 | □ | □ |  |  | □ |

**Object-oriented-programming (OOP)?**

**ECS = containers of components?**

# The map II (entity ID to component position)



ID Position

Mario 1

Goomba1 2

Luigi 3

Goomba2 4

ID Jumps

Mario 1

Luigi 3

Squishable

Goomba 1 2

Goomba 2 4

**Concept:** A (hierarchical) acceleration structure to lookup components

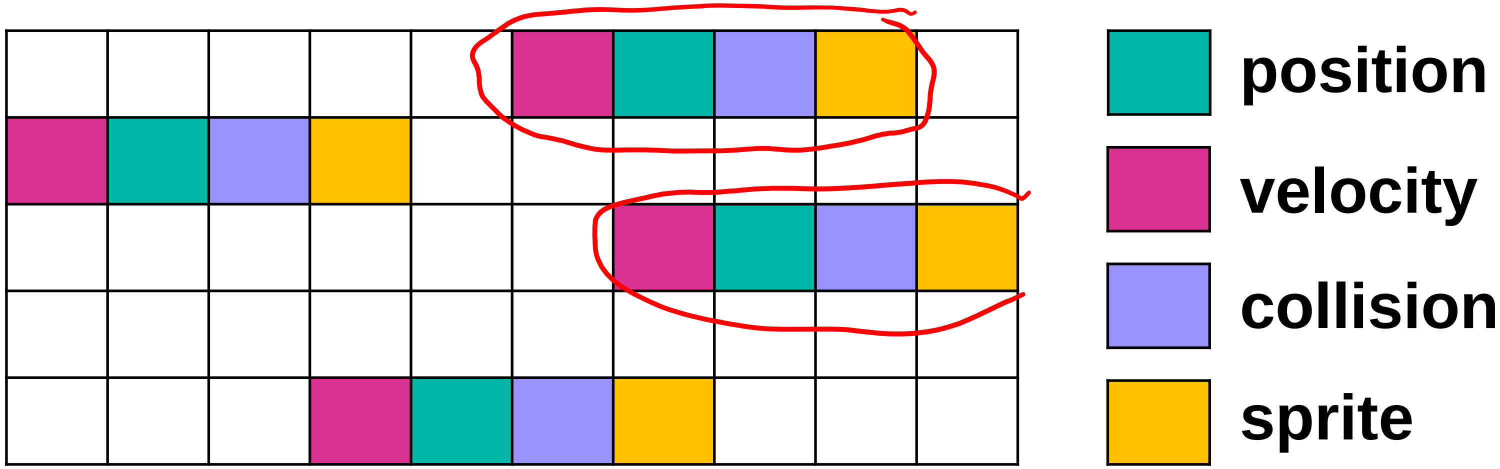
**Implementation:** std:map<Entity, Position>

© Alla Sheffer, Helge Rhodin

# Memory & ECS

## Where do we store our Components?

- Inside Entities?
- A map?



Memory Blocks

**Update loop has to access non-contiguous memory repeatedly!**

**Not memory efficient!**

# The (giant) Sparse Array

|  | ID | Position | Velocity | Jumps | Player | Squishable |
|---|---|---|---|---|---|---|
| Mario | 1 | □ | □ | □ | □ | x |
| Goomba1 | 2 | □ | □ | x | x | □ |
| Luigi | 3 | □ | □ | □ | x | x |
| Goomba2 | 4 | □ | □ | x | x | □ |

**Issues?**

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

**Implementation:** std:vector<Position>; std:vector<Velocity>

# The Bitset / Bitmap

|  | ID | Bitset/bitmap | Position | Velocity | Jumps | Player | Squishable |
|---|---|---|---|---|---|---|---|
| Mario | 1 | 11110 | □ | □ | □ | □ |  |
| Goomba1 | 2 | 11001 | □ | □ |  |  | □ |
| Luigi | 3 | 11100 | □ | □ | □ |  |  |
| Goomba2 | 4 | 11001 | □ | □ |  |  | □ |

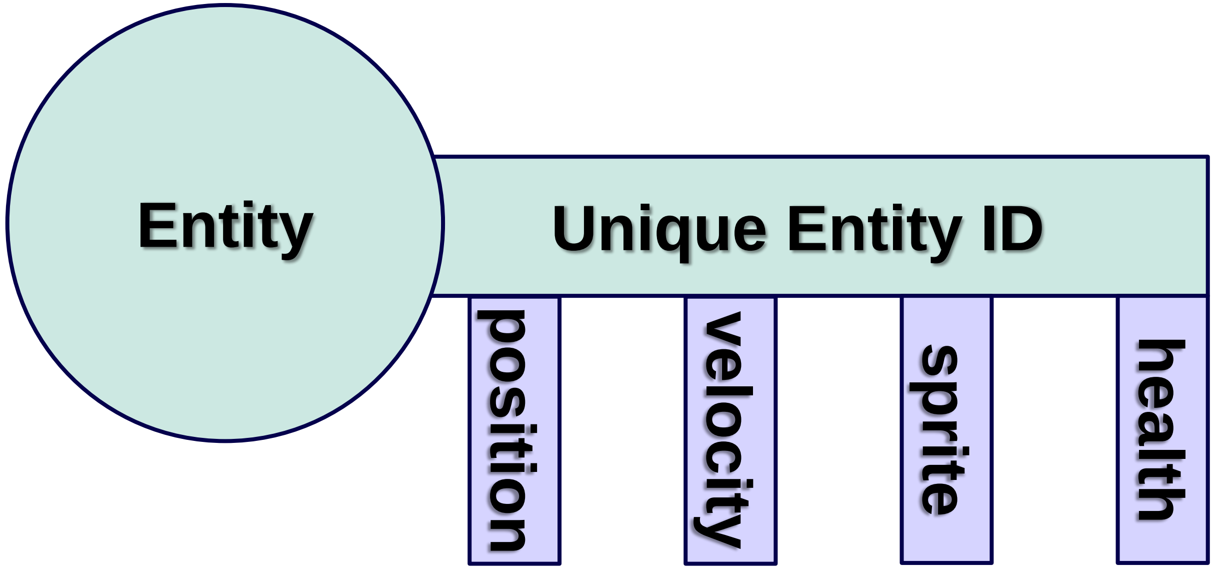
**Issues?**

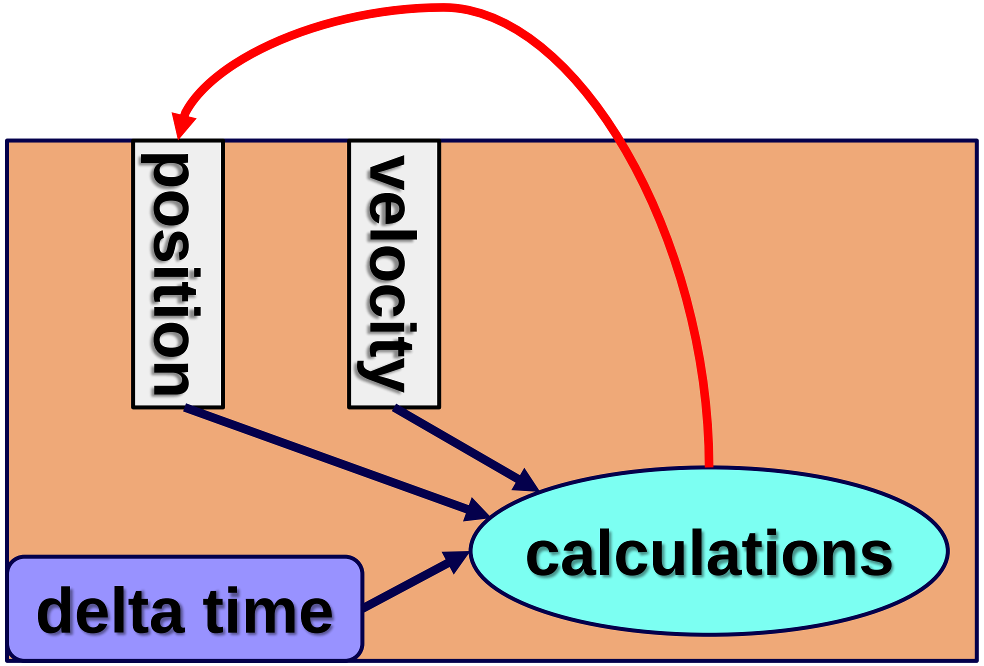
**Concept:** Each entity has a bitset that is true for its ‘owned’ components

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

If(bitset & query == query)   // has the entity all query components?

# Key & Lock Metaphor

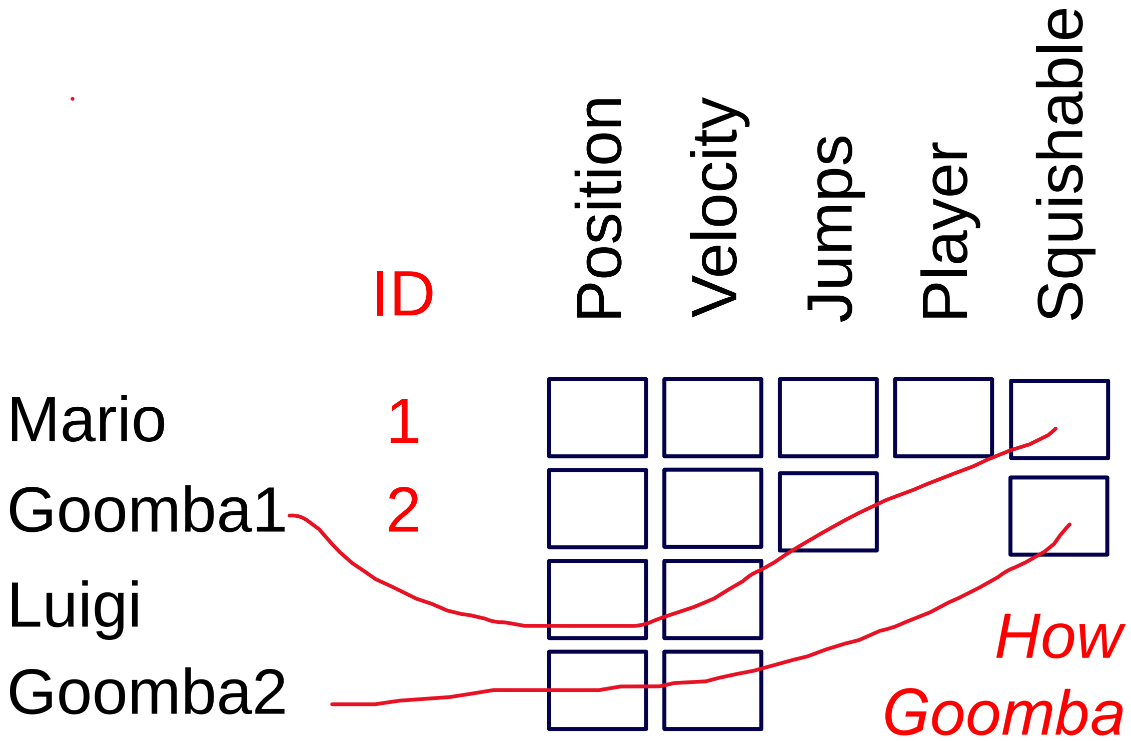




Motion System

Systems will only operate on Entities with the required Components

# The Dense Array (an attempt, needs more)



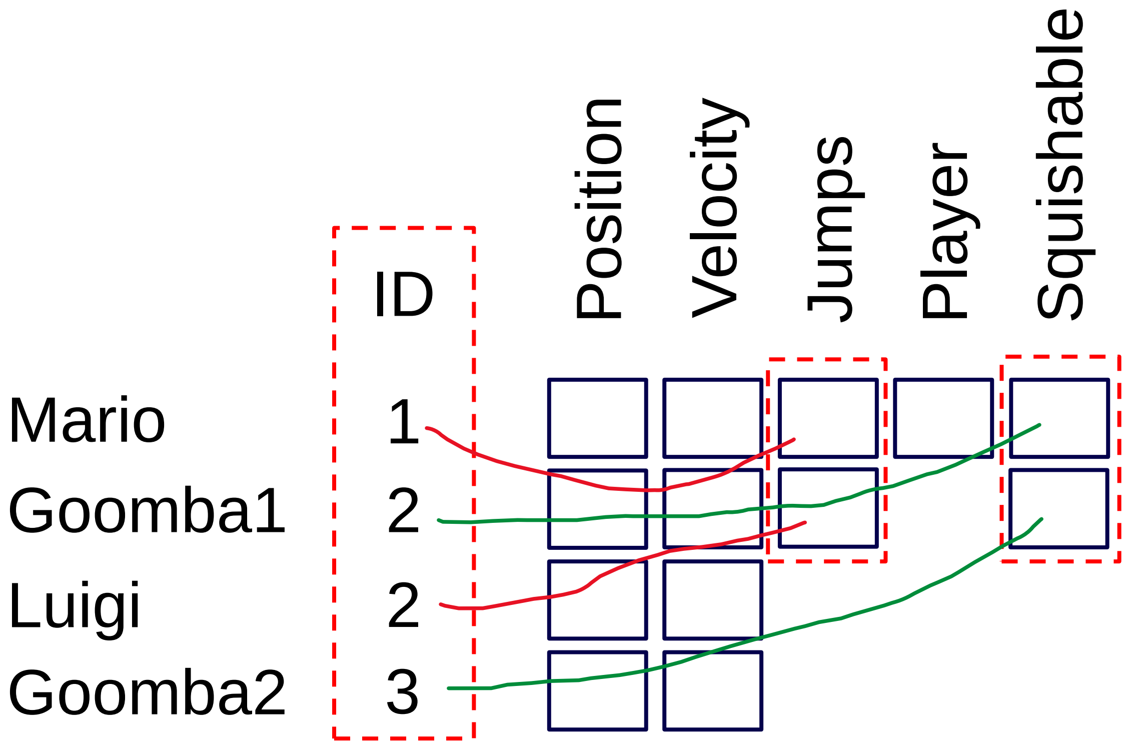
**Issues?**

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

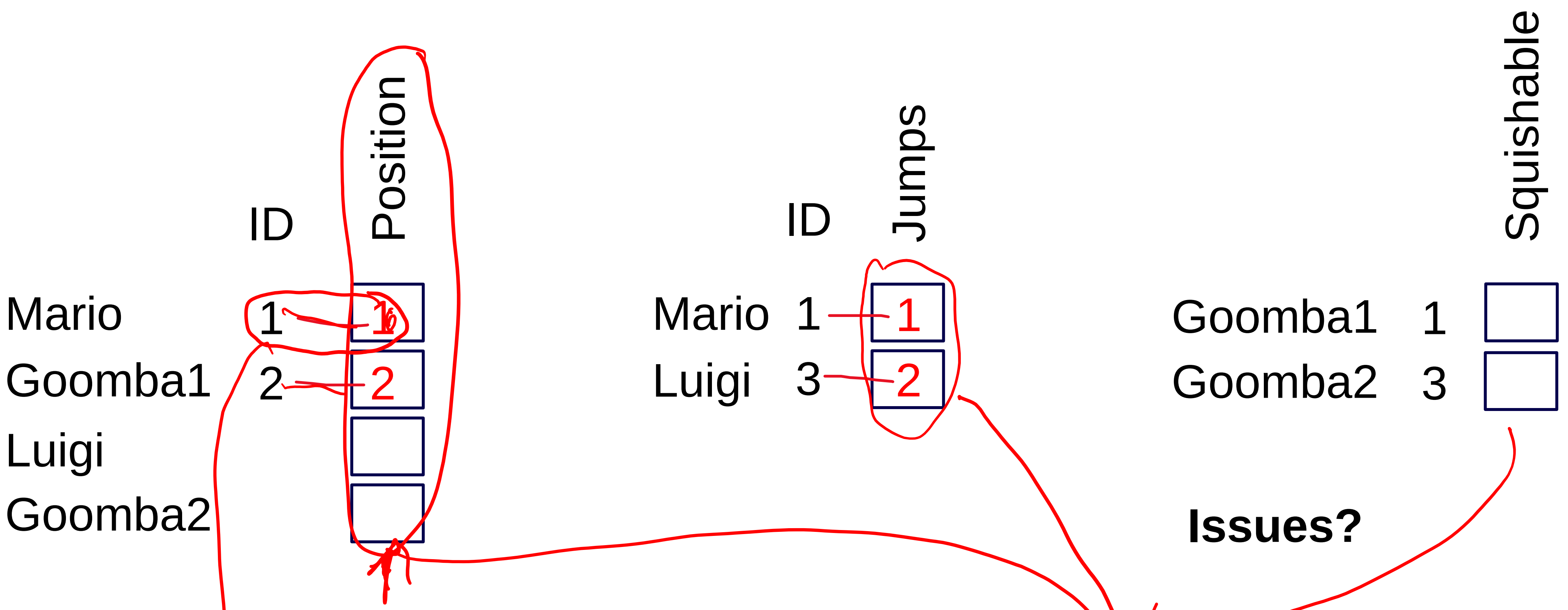
**Concept:** One array/vector per component, but how to associate?

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

# Map + Dense Array



# Map + Dense Array



| | ID | Position |
|---|---|---|
| Mario | 1 | 1 |
| Goomba1 | 2 | 2 |
| Luigi | | |
| Goomba2 | | |

| | ID | Jumps |
|---|---|---|
| Mario | 1 | 1 |
| Luigi | 3 | 2 |

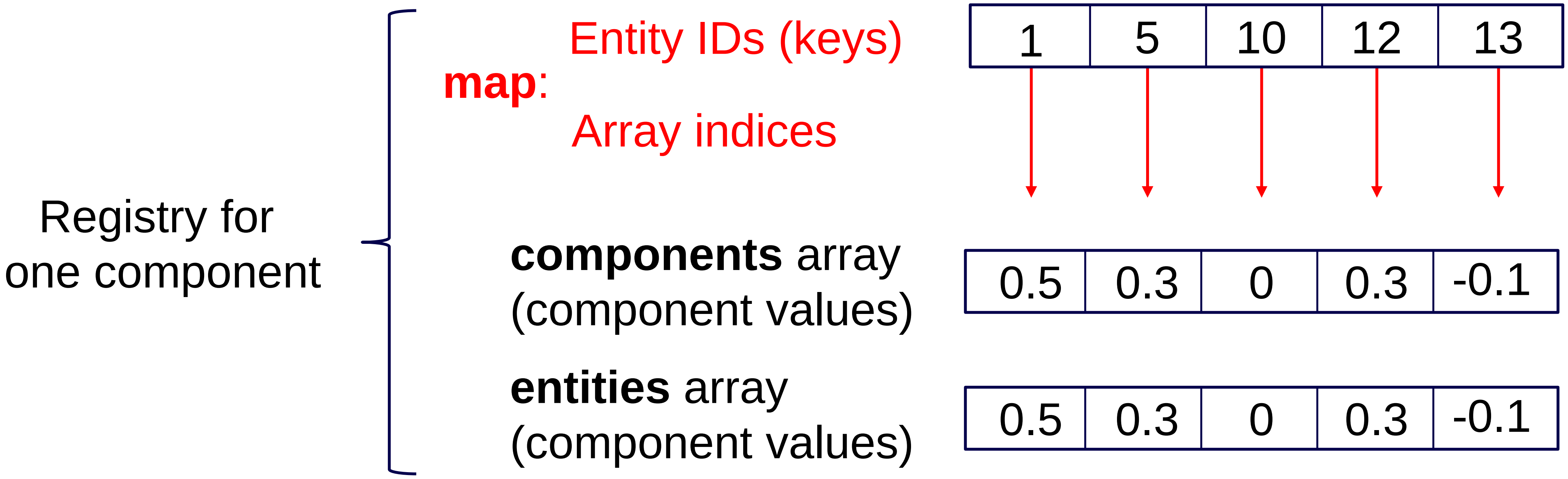
| | | Squishable |
|---|---|---|
| Goomba1 | 1 | |
| Goomba2 | 3 | |

**Issues?**

**Concept:** Combine dense arrays with a map

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

# Map + Dense Array (example)



Registry for one component

**map**: Entity IDs (keys) → Array indices

| 1 | 5 | 10 | 12 | 13 |
|---|---|---|---|---|

**components** array (component values)

| 0.5 | 0.3 | 0 | 0.3 | -0.1 |
|---|---|---|---|---|

**entities** array (component values)

| 0.5 | 0.3 | 0 | 0.3 | -0.1 |
|---|---|---|---|---|

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

```
for(Entity entity : registry<Velocity>.entities) // using the key array
    if (map<Position>.has(entity)) // using the map
        map<Position>.get(entity) += registry<Velocity>.get(entity); // using the map
```

# Faster iteration via entity and component array

Accessing the velocity map (map<Velocity>) is an unnecessary indirection

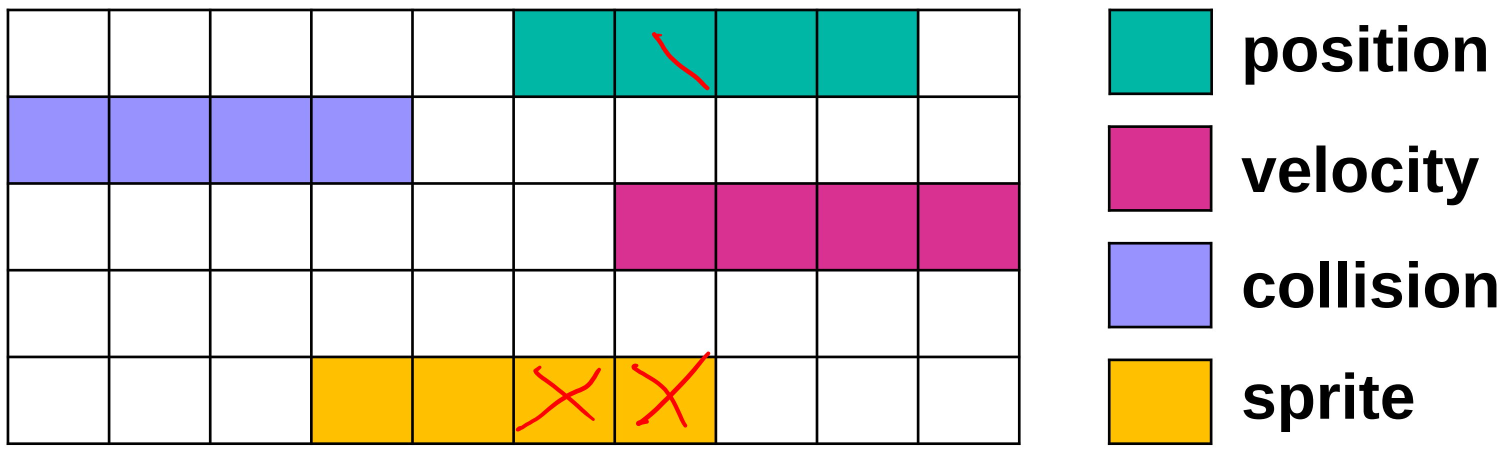
```
for(Entity entity : entities<Velocity>)
    if (map<Position>.has(entity))
        map< Position >.get(entity)+= map<Velocity>.get(entity);
```

We can access the velocity components in linear fashion

```
for(int vi = 0; vi < entities<Velocity>.size(); vi++)
    Entity entity : entities<Velocity>[vi];
    pi = map<Position>.get(entity);
    if (pi)
         components< Position >[pi]+= components< Velocity >[vi];
```

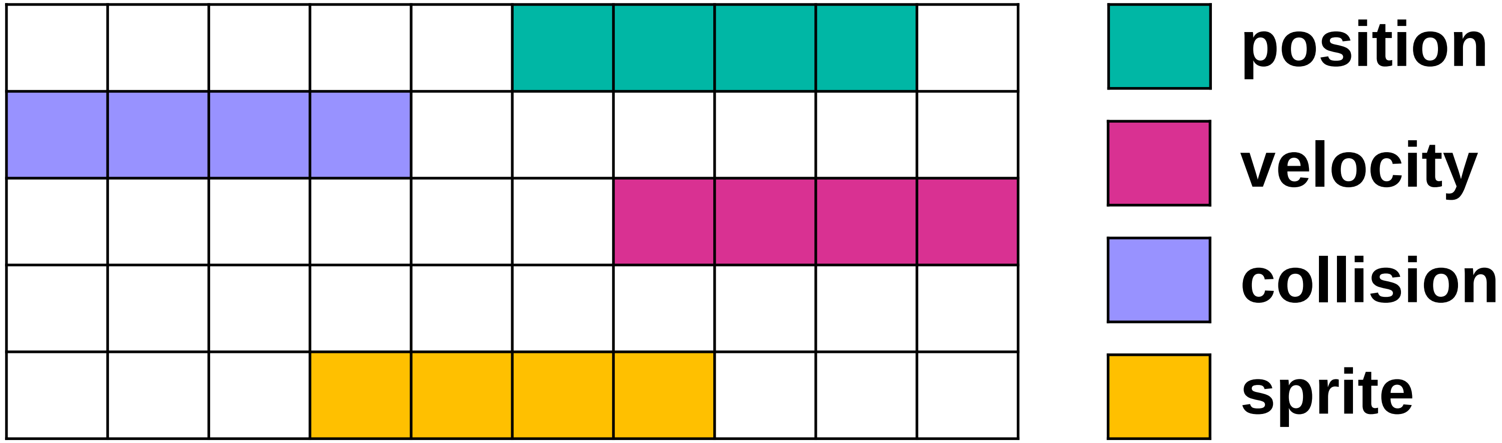
# Cache is Key

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



Memory Blocks

# Cache is Key



Memory Blocks

**Update loop accesses contiguous memory**

**IDEAL!**

# Convenient lookup with wrappers

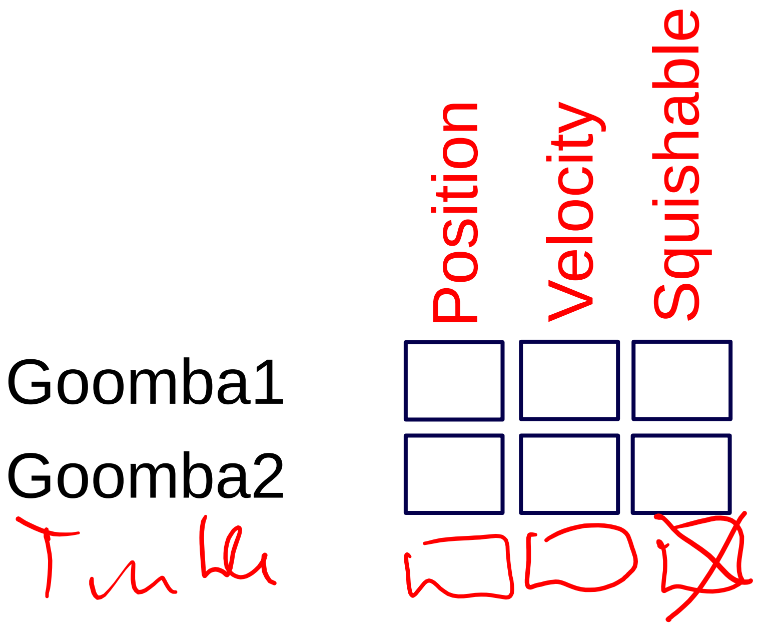
The user does not need to know about the internal storage in a map

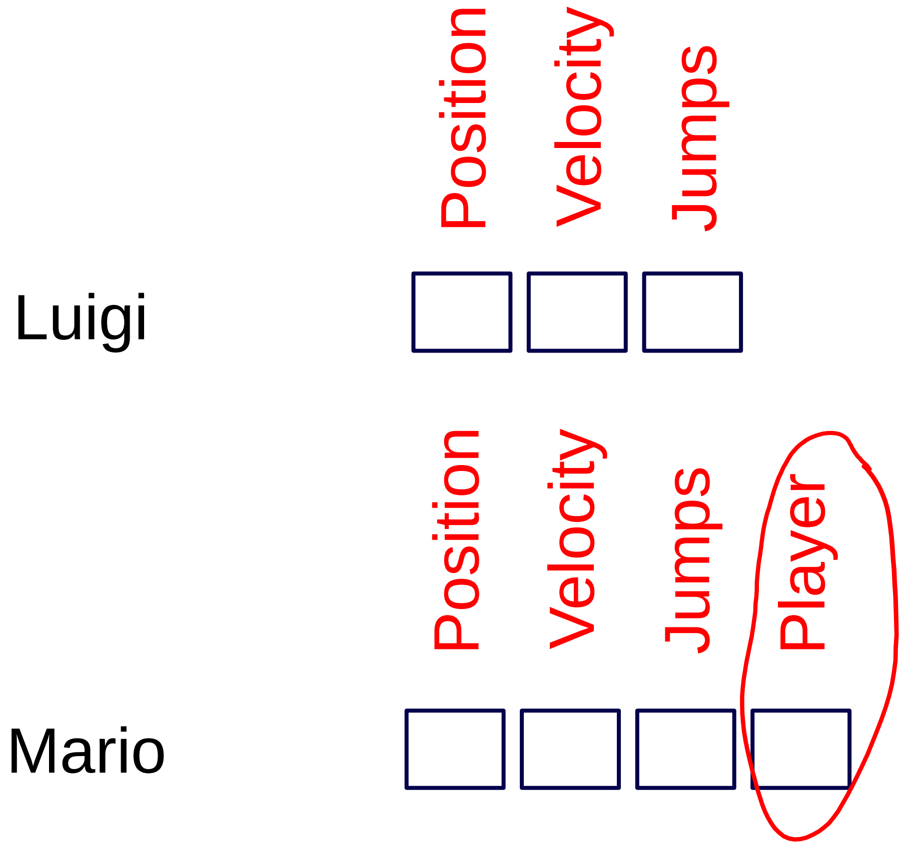
```
for(Entity entity : entities<Velocity>)
    if (map<Position>.has(entity))
          map< Position >.get(entity) += map<Velocity>.get(entity);
```

In A0 - Task 2, you define wrapper function to turn the one above into the one below

```
for(Entity entity : entities< Velocity >)
    if(entity.has< Position >() )
          entity.get<Position>() += entity.get<Velocity>()
```

# Archetypes / prototypes / pools





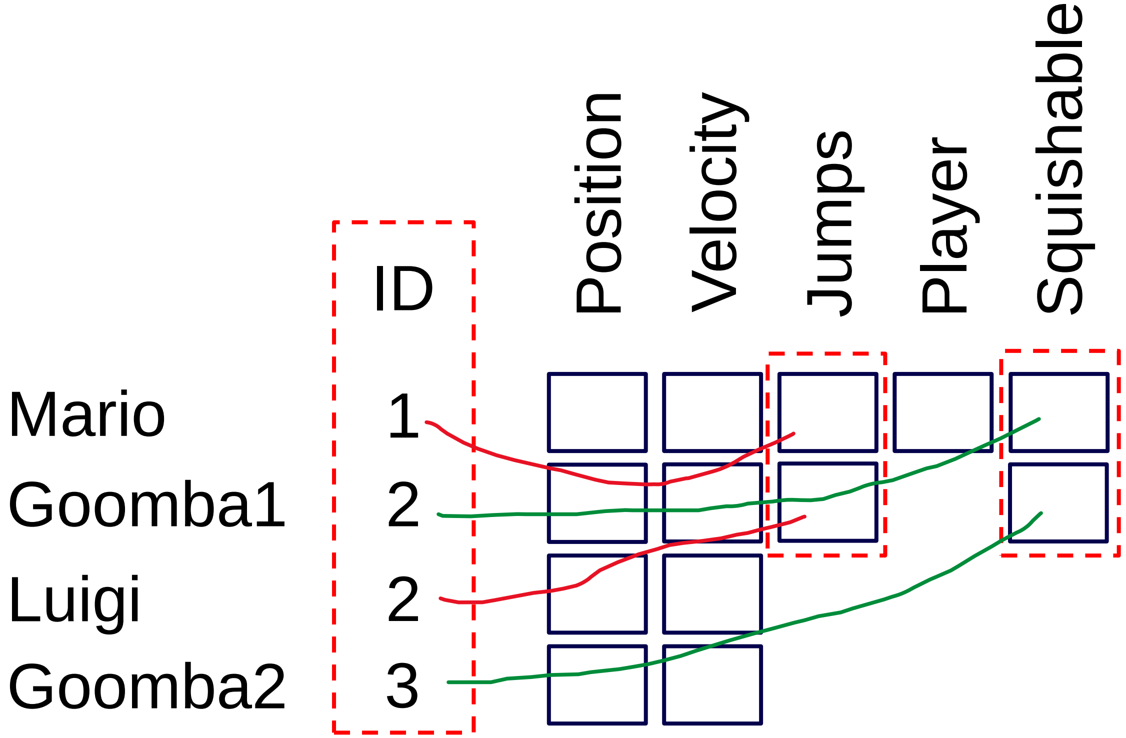
- **Concept:** store all types with the same components in dense arrays
- Used by the Unity ECS system
- Difficult to implement

# How Does a System Find its Entities?

## Extension: Entity Manager

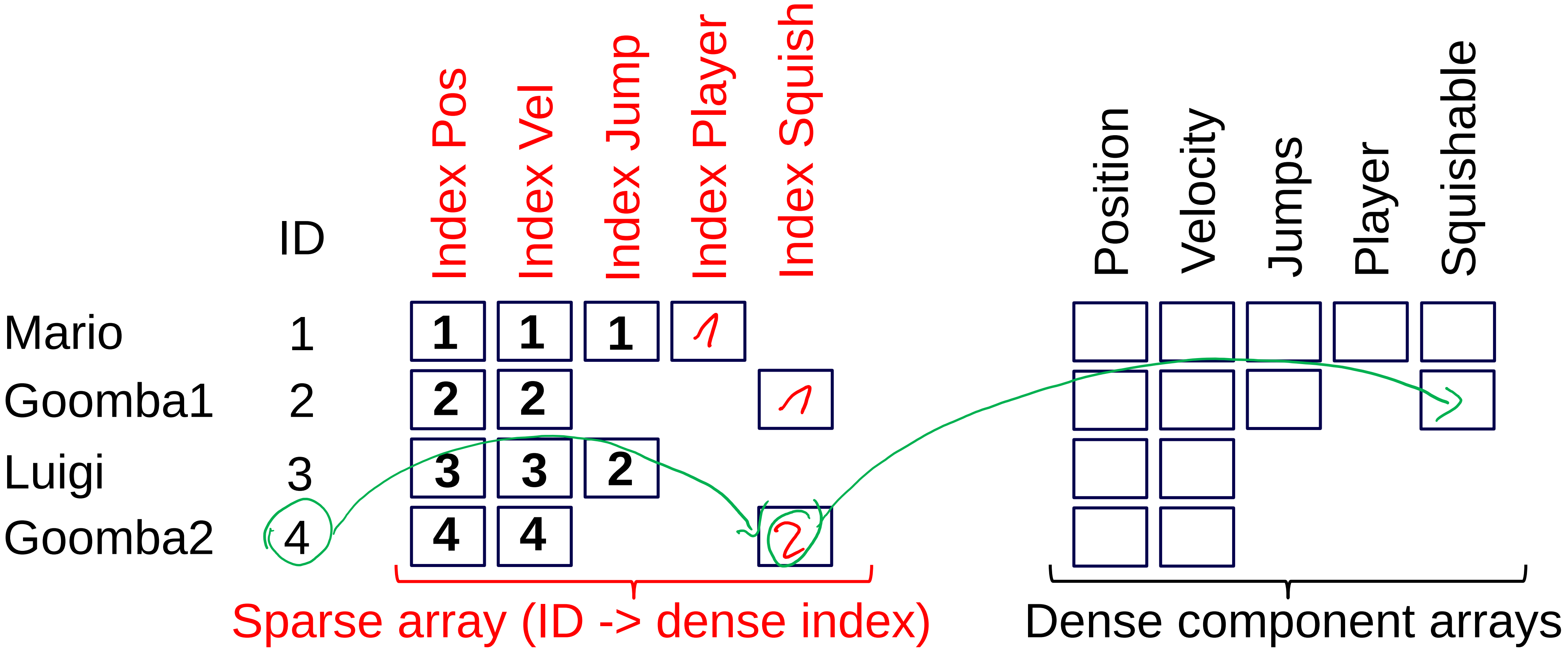
- Each system has a list of entity IDs it is interested in
- Systems register their bitsets/bitmaps with the Entity Manager
- Whenever an Entity is added…
  - *Evaluate which systems are interested & update their ID lists*

# Recap: the map approach



# The Sparse Map

***Used by*** ***https://github.com/skypjack/entt***



|  | ID | Index Pos | Index Vel | Index Jump | Index Player | Index Squish |
|---|---|---|---|---|---|---|
| Mario | 1 | 1 | 1 | 1 | 1 |  |
| Goomba1 | 2 | 2 | 2 |  |  | 1 |
| Luigi | 3 | 3 | 3 | 2 |  |  |
| Goomba2 | 4 | 4 | 4 |  |  | 2 |

Sparse array (ID -> dense index)

Dense component arrays

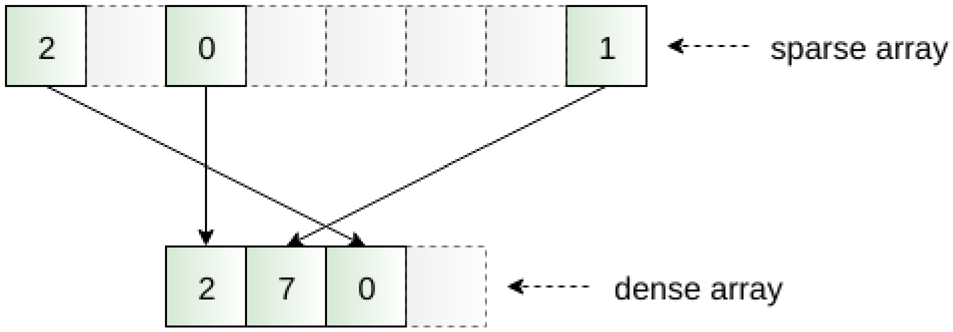
**Issues?**

**Concept:** Sparse array + dense array

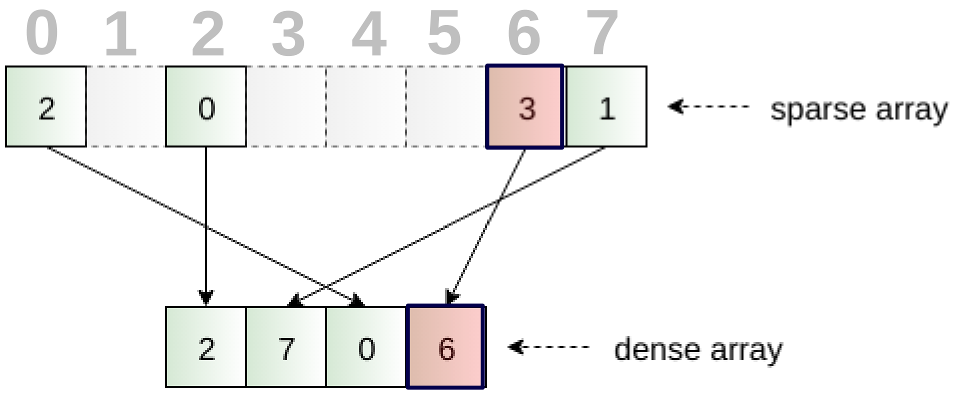
**Implementation:** std:vector<Entity> entities; std:vector<unsigned int> indices; std:vector<Components> components;

# Faster Lookup with Sparse Sets

**Lookup:**



**Insert:**



[https://skypjack.github.io/2020-08-02-ecs-baf-part-9/]

The map lookup (map<Velocity>.get(entity)) is costly

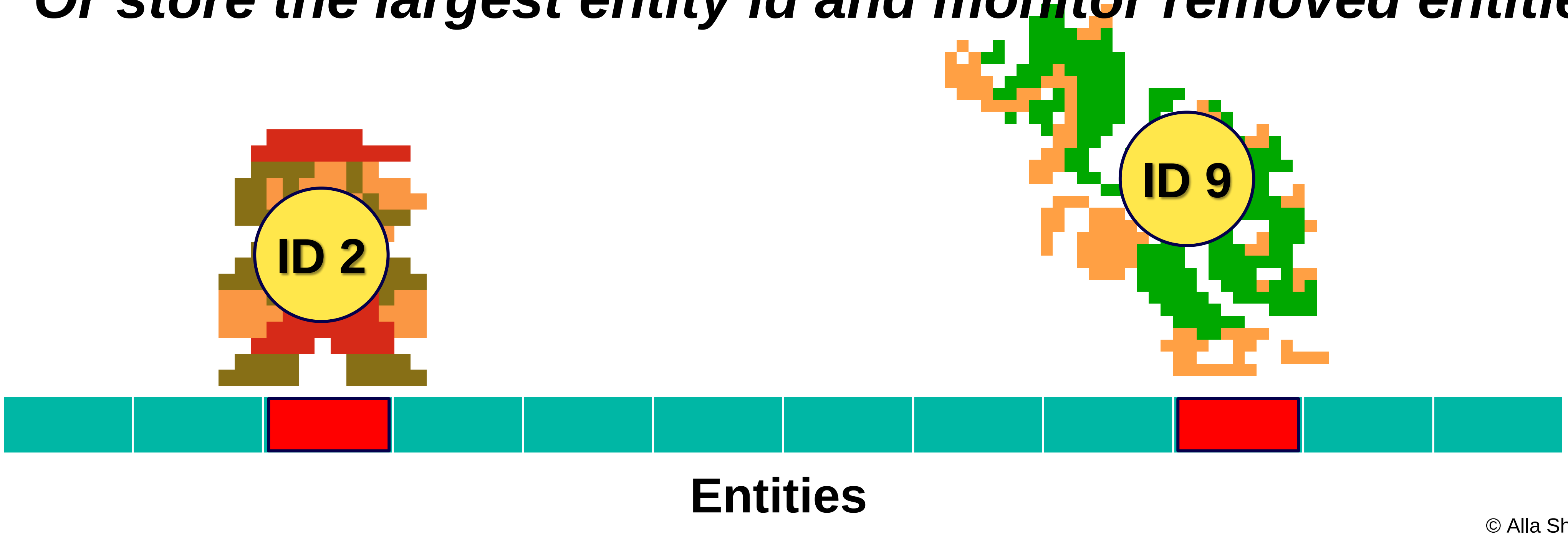
- A hashmap is O(1), but 1 can be big

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
  - *Or store the largest entity id and 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!**

© Alla Sheffer, Helge Rhodin

# Deletion of components

- **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

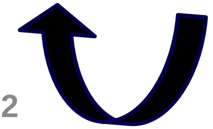
## Can you imagine a game without?

# A game is a simulator

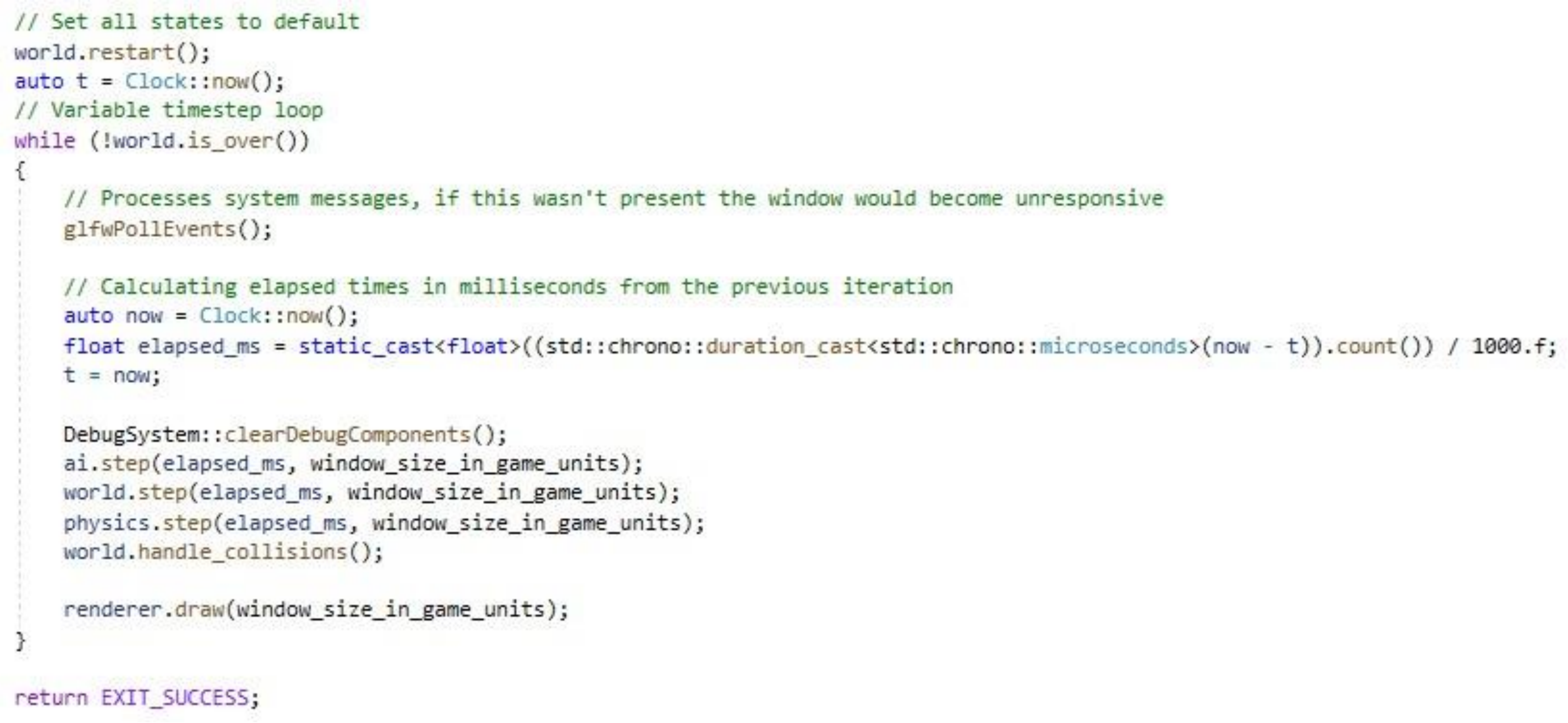
1. ***AI and user input***
2. ***Environment reaction***
3. ***Equations of Motion***
   - sum forces & torques, solve for accelerations: $\boldsymbol{\vec{F} = ma}$
4. ***Numerical integration***
   - update positions, velocities
5. ***Collision detection***
6. ***Collision resolution***

← *Also simulation forms!*

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



# 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;
```