CS354R

DR SARAH ABRAHAM

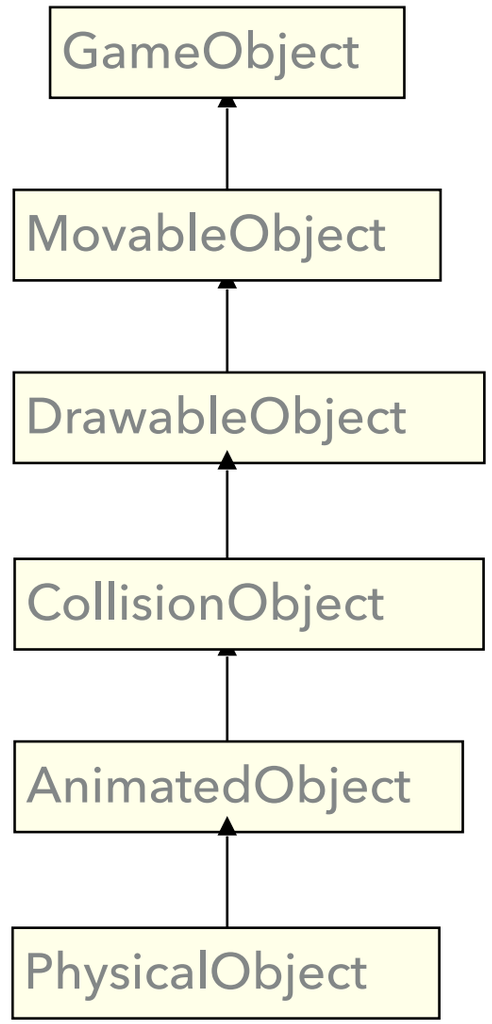
COMPONENT-BASED SOFTWARE DESIGN

GAME ENGINE ARCHITECTURE

- Good practices lay a good foundation

INHERITANCE-BASED ARCHITECTURE

- Deadly diamond
- Hard to maintain
- Messy structure
- Potential memory penalties

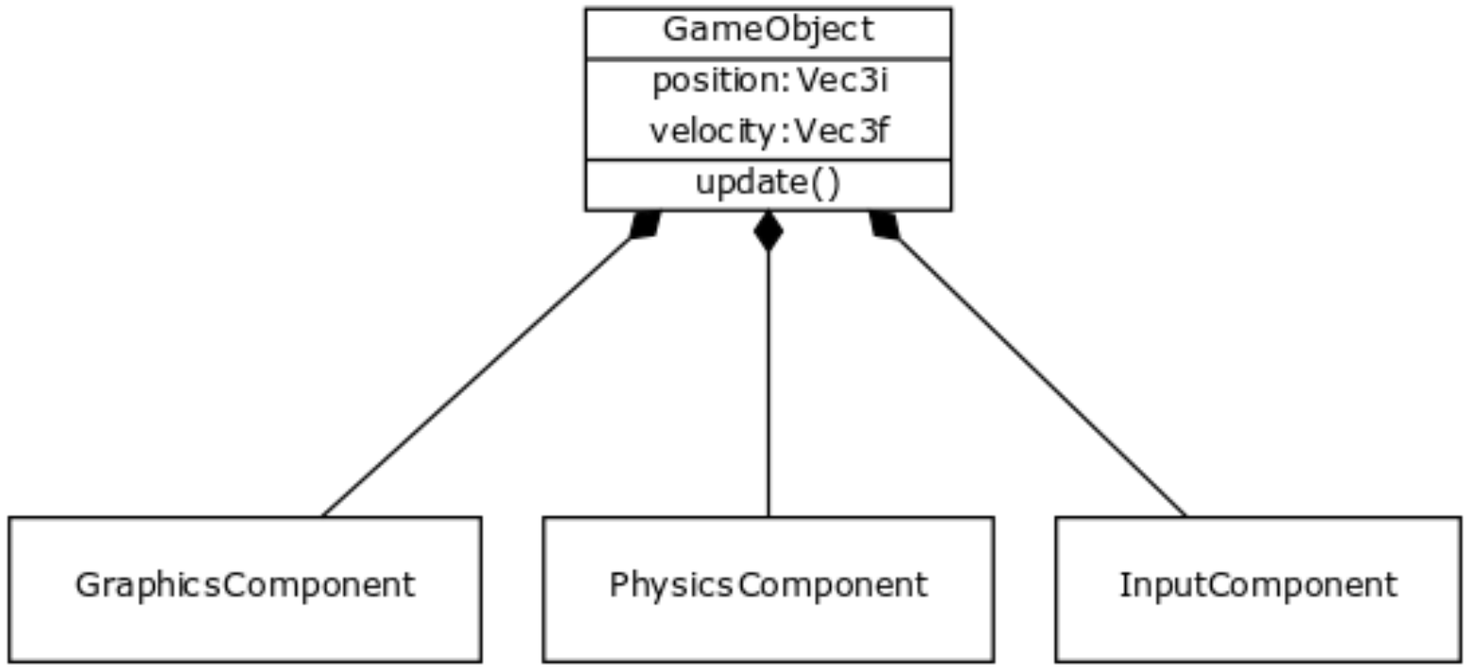


PROBLEM

- Entities inherently occupy multiple domains
- Domains should remain as isolated as possible

COMPONENT-BASED ARCHITECTURE

- Break domains into component classes
- Entity acts as a container of components

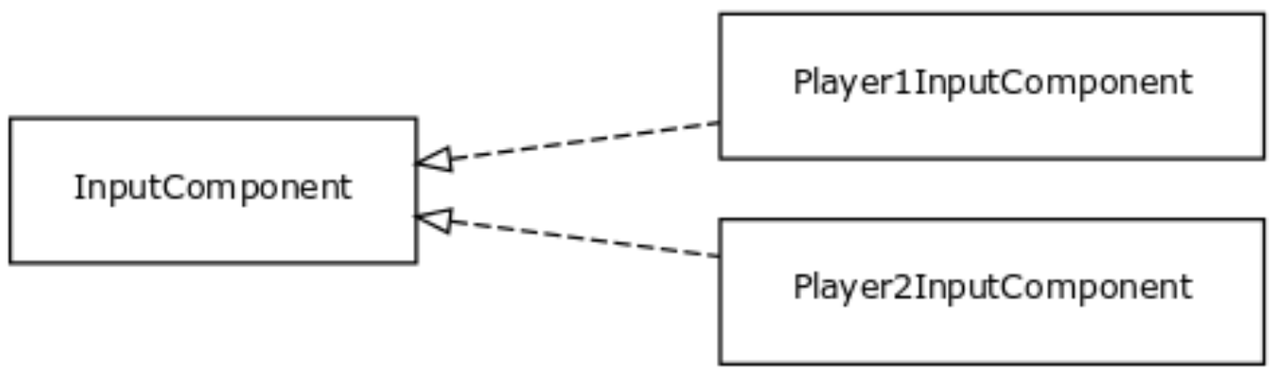


POTENTIAL DOMAINS

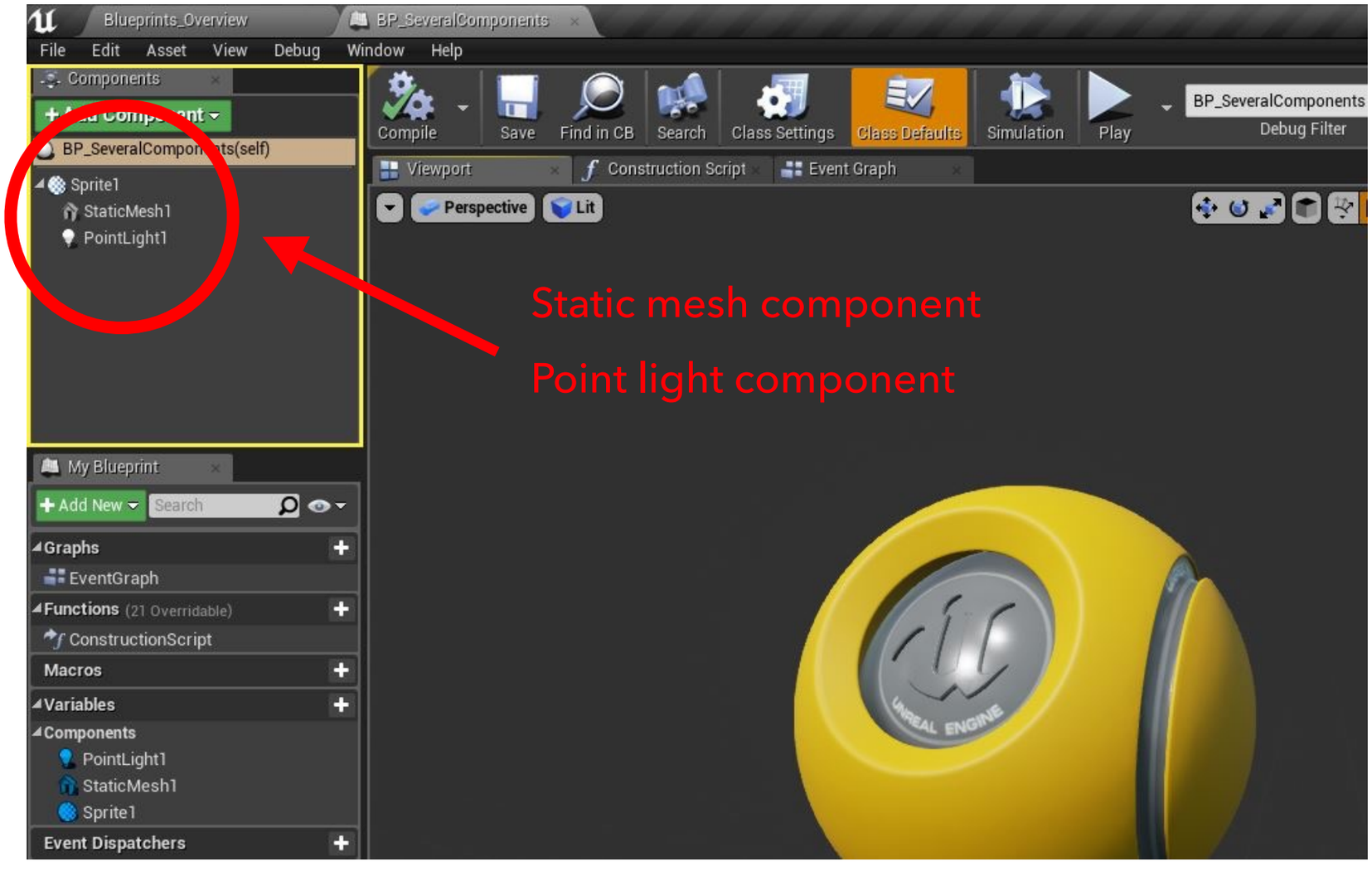
- Domains can be a single component or broken into multiple components:
  - Input
  - Graphics (Rendering + Animations)
  - Physics (Collision + Forces)
  - Sound
  - GUI
  - AI (Sensing + Thinking + Acting)
  - Game Logic

ABSTRACT BASE CLASSES

- Components as abstract base classes implemented via interfaces
  - System knows when/where to call methods
- User implements specific functionality needed by parent object



COMPONENTS IN UNREAL

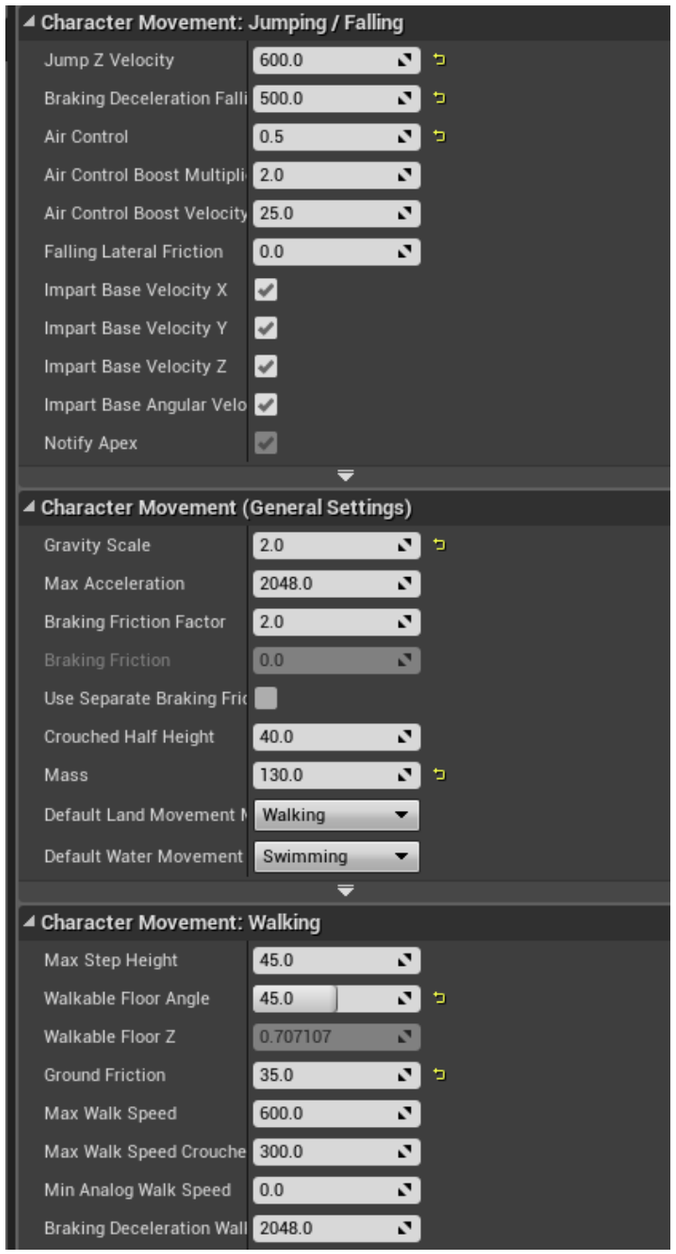


UACTORCOMPONENTS

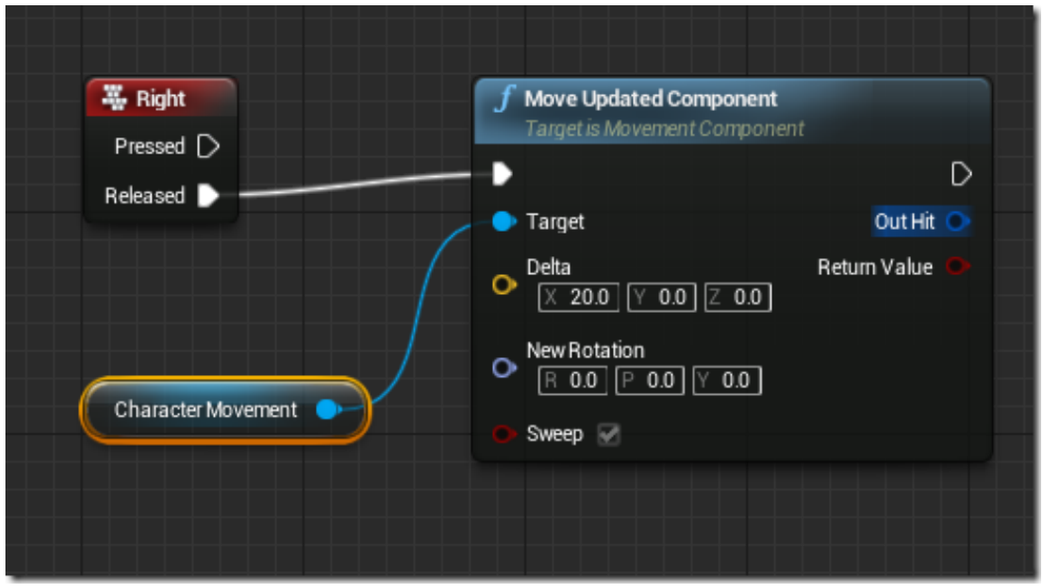
- Only way to render meshes, implement collision, play audio, etc
- Scene components have location but no geometric representation
- Primitive components have location and geometric representation
- To interact with larger systems (e.g. rendering, physics, etc), components must be given **state** for that system
  - Provides component with properties system requires
  - Allows for more efficient updating when state is “dirty”

COMPONENT PROPERTIES

- Character Movement Component controls all movement associated with Character objects
  - Walking, Jumping/Falling, Swimming, Flying etc
- Physics calculations and networking replication handled within the Character Movement Component

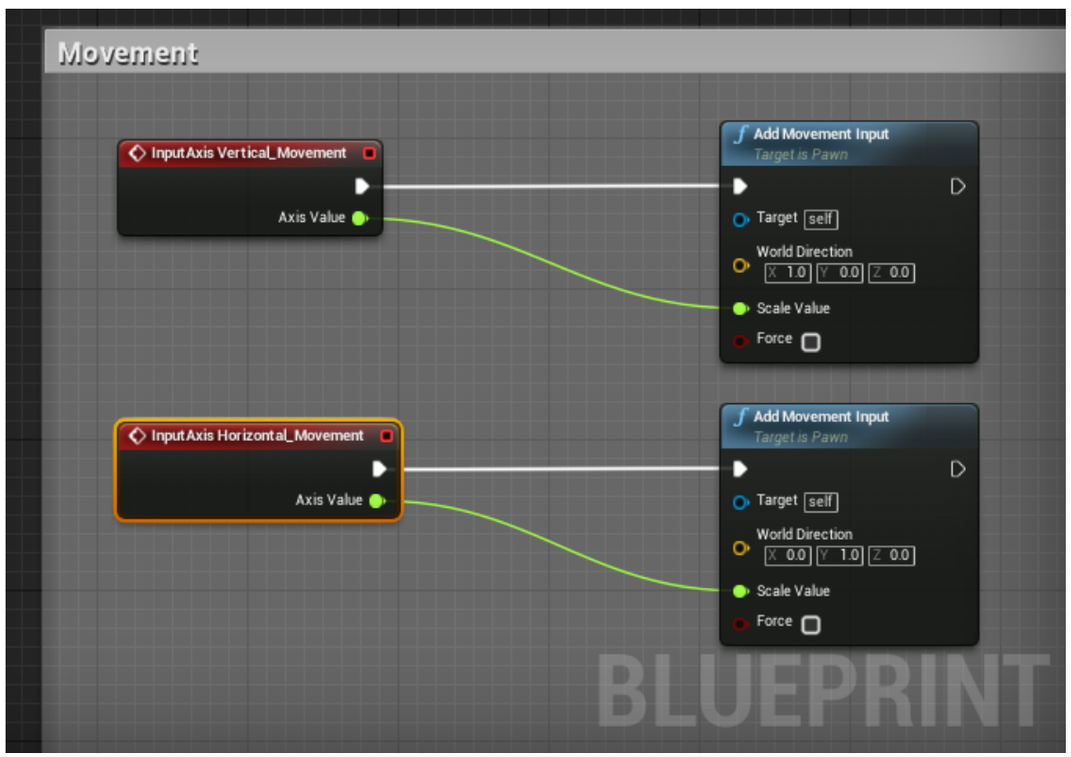


CONNECTING VIA BLUEPRINT



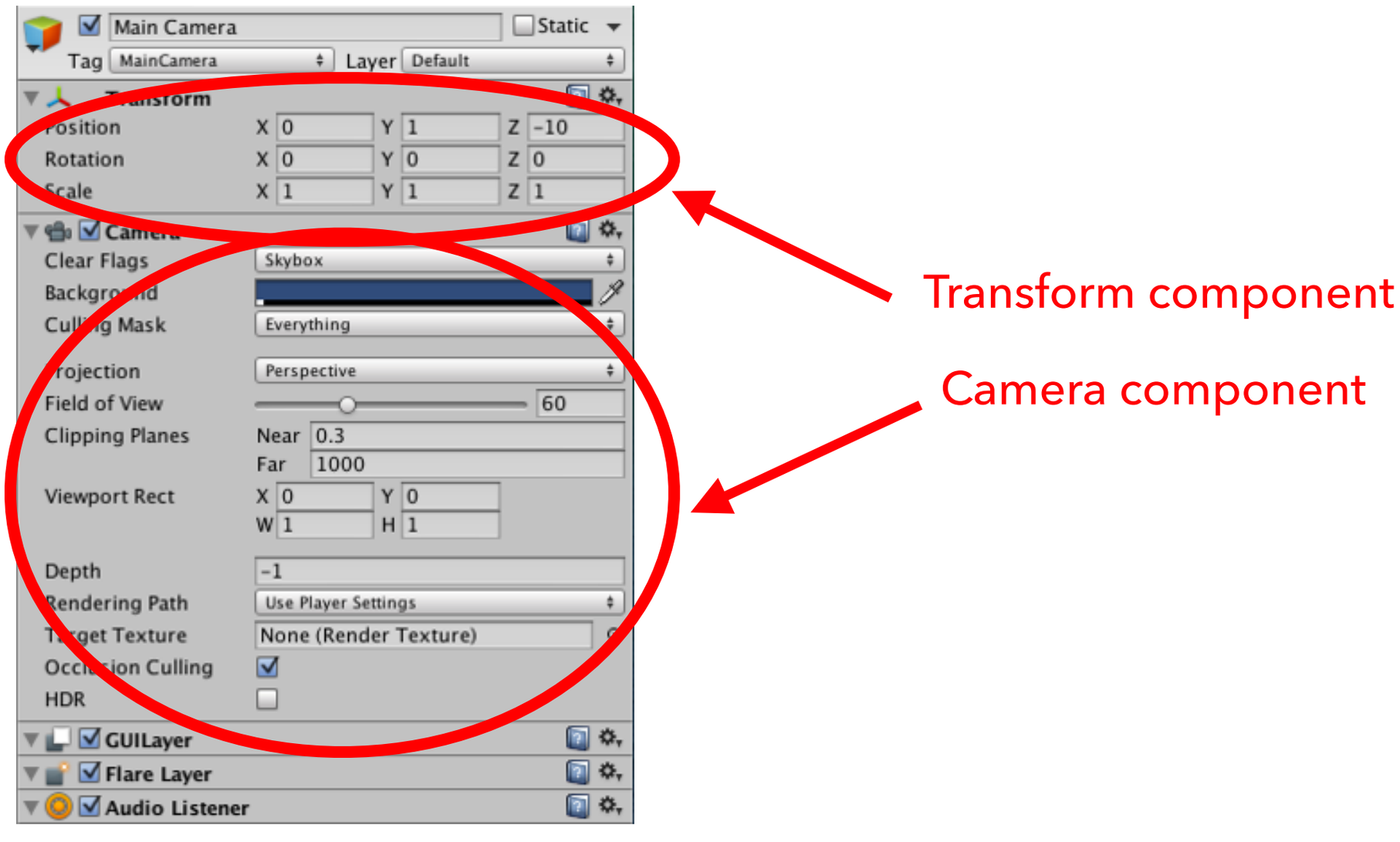
Directly access the component

Access component through parent



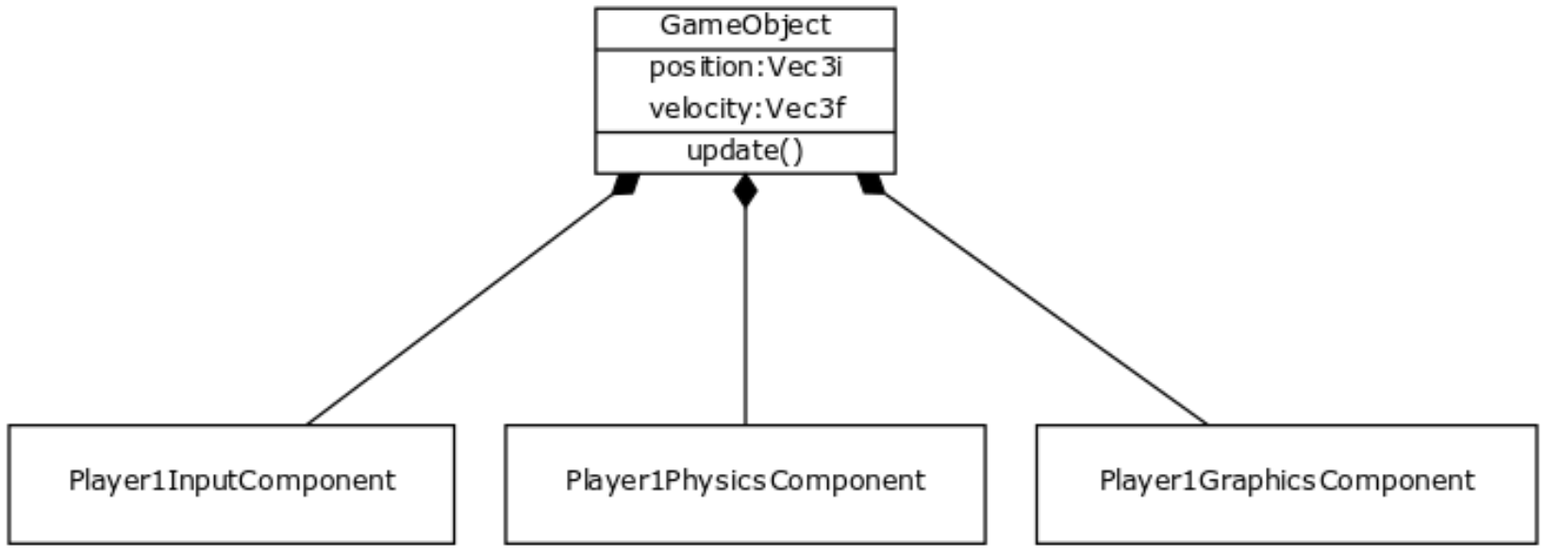
COMPONENTS IN UNITY

Main Camera Game Object associated with each scene



GAMEOBJECTS WITHOUT INHERITANCE

- No need for GameObject inheritance
- Instantiate GameObjects based on selected components:



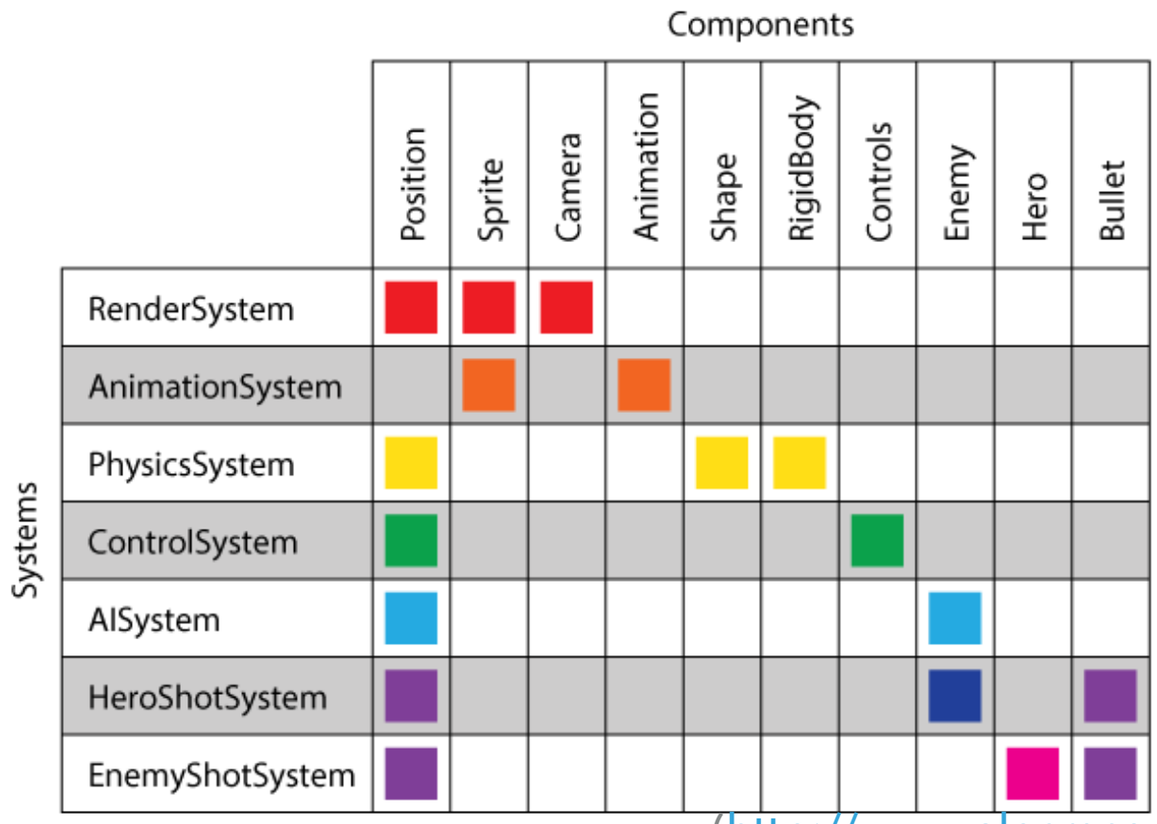

```
GameObject * createPlayer1() {  return new GameObject(new
Player1InputComponent(),  new Player1PhysicsComponent(),
new Player1GraphicsComponent()); }
```

PURE COMPONENT-BASED DESIGN

- If we take this model of container classes with components to its extreme, we get Entity-Component-Systems

ENTITY COMPONENT SYSTEM

- A specific form of component-based architecture
  - Entity is an id
  - Entity data stored as components
  - Systems modify related components



(http://www.alecmce.com/)

How do these systems communicate? (e.g. how can an entity update its sprite during a collision?)

COMPONENT COMMUNICATION

- Direct Reference
  - Components have references to relevant components
  - Example:
    - GraphicsComponent contains a reference to PhysicsComponent
    - Updates sprite upon collision
- Message Passing
  - Component sends message to container class
  - Container class broadcasts message to its components
  - Same idea as event-driven programming

IMPLEMENTING ECS

- Components grouped by an ID form a “game object”
  - Need fast component lookup by ID
- Factory classes create components for each game object type
  - Alternatively, “data-driven” model can read in a file defining object types
- Inter-object communication requires sending a message to an “object” to get required response
  - Know a priori which component gets a given message
  - Multicast to all of the components of an object

RESTRUCTURING THE ENGINE LOOP

- GameObjects contained in vector at game manager level

```
class Engine {

    std::vector<GameObject>;

    void update();

}
```

- GameObject components stored in vector within component managers:

```
class PhysicsManager : public Manager {

    std::vector<PhysicsComponent>;

}
```

STORING DATA

- Game Objects are a unique id:

```
struct GameObject {
    unsigned int id;
};
```

- Components contain relevant data:

```
struct PhysicsComponent {
    vector3 position;
    quaterion orientation;
    vector3 velocity;
};
```

UPDATING THE SYSTEM

- System accesses entities with relevant components during update loop

```
class PhysicsSystem : public System {

    std::vector<GameObject> entities;

    void update() {

        for (entity in entities) {

            physics = getPhysicsComponent(entity);

            physics.position += physics.velocity;

        }

        ...
```

Systems add and remove entities as their components change

CREATION AND DESTRUCTION

- What do we need to consider when creating or destroying objects in an entity-component system?

MANAGING MEMORY

- Essential for entity-component systems
- Smart layout of data will avoid cache misses
- Cache hits lead to massive performance gains
- Arrays are flat with fast access
- Vectors allow for flexibility in array size

ECS PROS

- Can be more memory-efficient
  - Only store properties in use, no unused data members in objects
- Easier to construct in a data-driven way
  - Define new attributes with scripts, less recoding of class definitions
- Can be more cache-friendly
  - Data tables loaded into contiguous locations in cache
  - Struct of arrays (rather than array of structs) principle

ECS CONS

- Hard to enforce relationships among properties
- Harder to implement large-scale behaviors if they’re composed of scattered pieces of fine-grained behavior
- Harder to debug
  - Can’t just put a game object into a debugger watch window and see what happens to it

HYBRID SOLUTIONS POSSIBLE

- Hierarchies are messy, but component-based systems might be over-engineering
- Always design for the problem
- Usual software principles:
  - Take time to plan before writing code
  - If a system is difficult to conceptualize, the current approach might be wrong
  - **Leave time to rework existing code**

LAST THOUGHTS ON DATA STRUCTURES

- **There is no one correct solution**
- Individual preference is a good place to start…
  - But be flexible and adapt to the problem
- Don’t over-engineer or prematurely optimize…
  - But keep data storage and caching in mind
- Try different approaches
- Take multiple passes to refactor

REFERENCES

- Scott Bilas. A Data-Driven Game Object System<http://scottbilas.com/files/2002/gdc_san_jose/game_objects_slides.pdf>
- Bob Nystrom. Game Programming Patterns <http://gameprogrammingpatterns.com/component.html>
- Randy Gaul. Component Based Engine Design <http://www.randygaul.net/2013/05/20/component-based-engine-design/>
- Nomad Game Engine <https://medium.com/@savas/nomad-game-engine-part-2-ecs-9132829188e5>