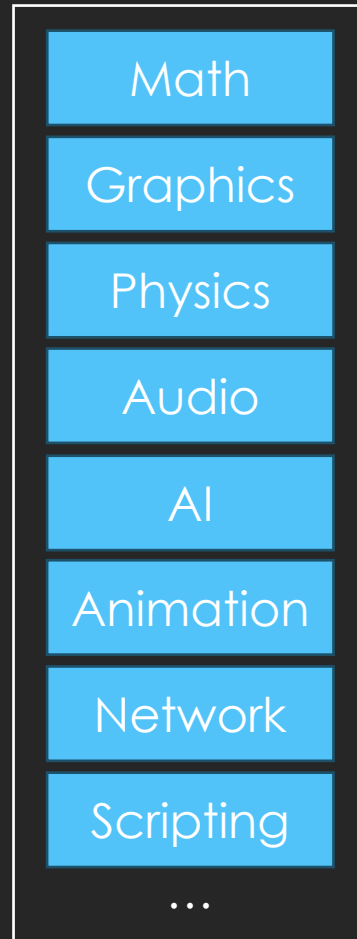


# Data organization in the assignment

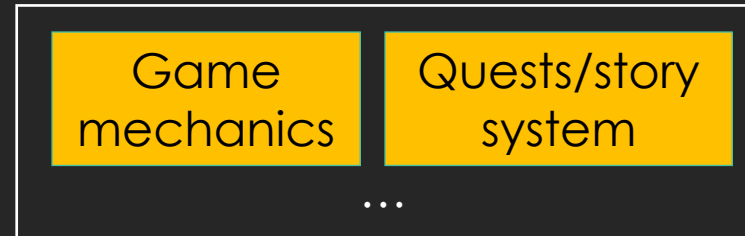
Marek Trtík  
PA199

# Typical game architecture

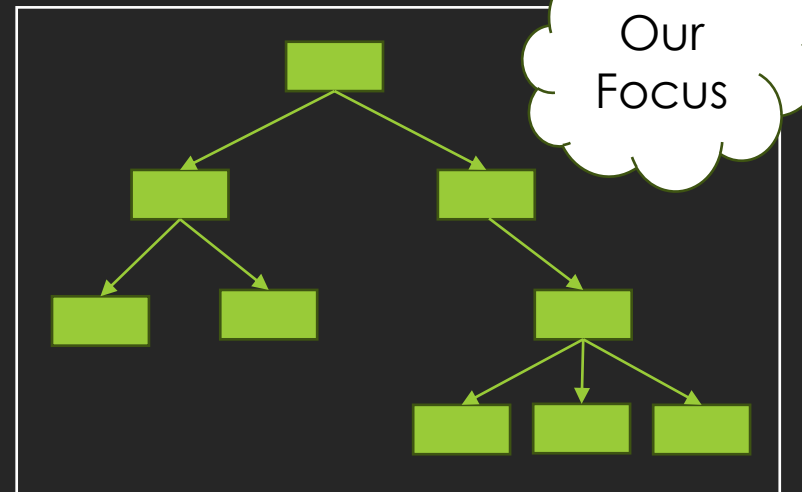
Engine libraries



Game libraries



Game data



# Game data: Tree structure

- ▶ General enough for games:
  - ▶ Game objects are often composed in an acyclic hierarchies, e.g.:
    - ▶ Car – the wheels and doors are attached to the body.
    - ▶ Animation skeleton – bones form a tree.
    - ▶ Items (like weapons) are attached to an agent/player.
- ▶ Simple to implement and easy to traverse (tree DFS, BFS):

```
class GameNode {  
    GameNodeWeakPtr parent;  
    std::list<GameNodePtr> children;  
    ...  
}
```

# Game data: Tree structure

```
class GameNodeHierarchy { // The only owner of all nodes
    GameNodePtr root;
```

```
public:
```

```
    static GameNodeHierarchy& instance(); // Singleton
```

```
template<typename NodeType, class... ParameterTypes>
```

```
std::shared_ptr<NodeType> push_back_child(
```

```
    GameNodePtr parent,
```

```
    ParameterTypes... args);
```

```
...
```

# Game data: Component based

- ▶ A **component** is a class instance of some Engine/Game library.
  - ▶ Holds specific data.
  - ▶ May also provide a functionality – code.
- ▶ Allows for a **data-driven** approach:
  - ▶ Attaching components to a GameNode => Specification of node's purpose in the game.
  - ▶ Intuitive and easy to use.
- ▶ Also easy to implement:

# Component system

```
class Component {  
    bool active;  
    GameNode* node; // The node this component is attached to.  
public:  
    virtual ~Component() {} // IMPORTANT: Allows for an inheritance!  
    ...  
};
```

- ▶ We need to extend GameNode to store attached components:

# Component system

```
class GameNode {  
    ...  
    std::list<ComponentPtr> components;  
public:  
    template<typename ComponentType>  
    std::shared_ptr<ComponentType> find_component() const;  
    void push_back_component(ComponentPtr component);  
    ...  
};
```

- ▶ An important component for a game is a **frame of reference**:

# Frame of reference

```
class Frame : public Component { // Frame of reference
    FramePtr parent; // A frame in which this one is defined.
                    // We do not need to know about children.

    Vec3 origin;
    Quat orientation;
    Vec3 scale; // Optional; for uniform scaling use just: float scale;

    mutable Mat44* to_world; // Cached; compute on demand.
    mutable Mat44* from_world; // Cached; compute on demand.
    ...
}
```



# Scripting in C++

- ▶ We can define a script component:

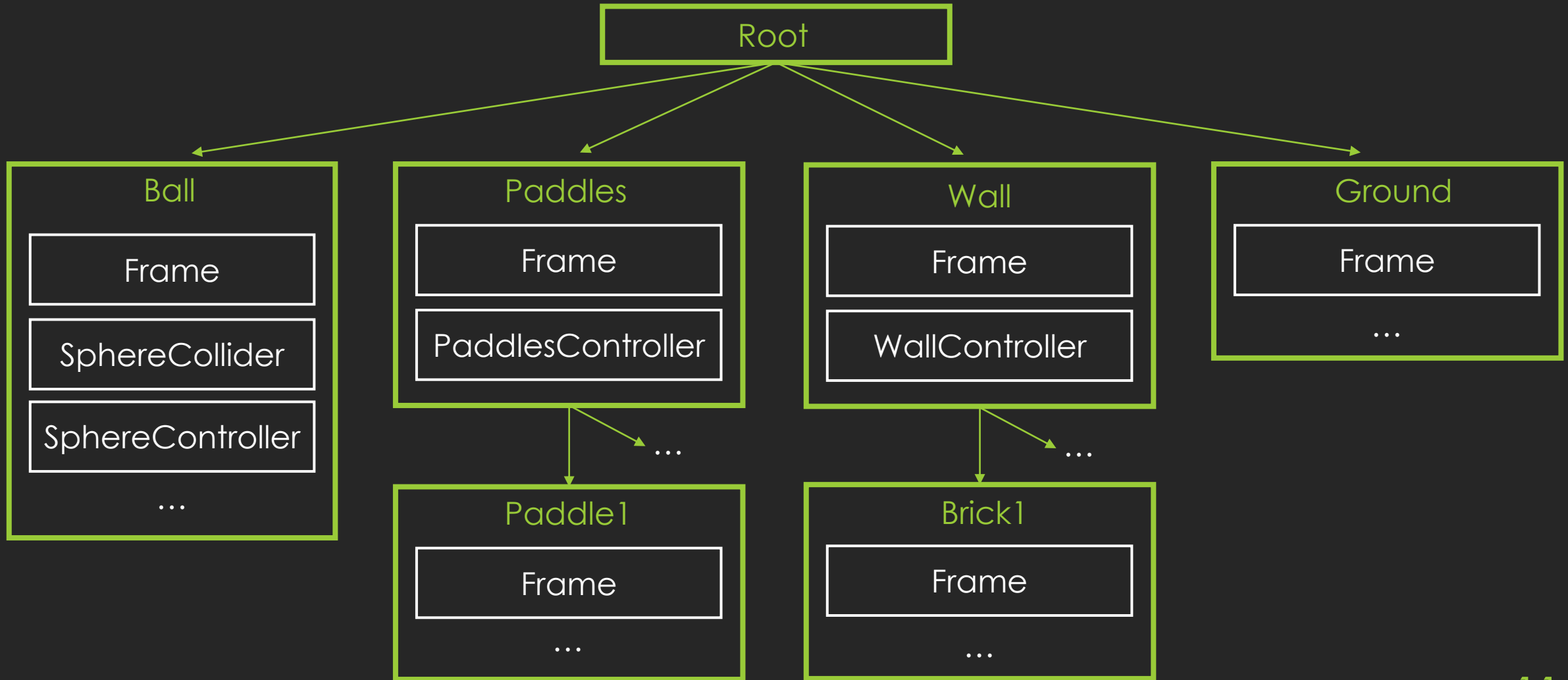
```
class GameScript : public Component {  
public:  
    virtual void update() {} // To be called by ScriptingEngine  
};
```

- ▶ A simple scripting engine can then be defined as follows:

# Scripting in C++

```
class ScriptingEngine {  
    std::list<GameScriptPtr> scripts;  
public:  
    static ScriptingEngine& instance(); // Singleton  
    void update(); // Call 'update' on each script.  
    template<typename ScriptType, class... ParameterTypes>  
    std::shared_ptr<ScriptType> create_script(ParameterTypes... args) {  
        auto script = std::make_shared<ScriptType>(args...);  
        scripts.push_back(script); // Keep track of all created scripts.  
        return script;  
    }  
    ...  
};
```

# Example: Data hierarchy of our game



# Reference

- ▶ A sketch of an implementation of the discussed topic is in IS:  
game\_data\_hierarchy.ZIP