Game Programming Paradigms

Michael Chung
CS248, 10 years ago...
Goals
Goals

1. High level tips for your project’s game architecture
Goals

1. High level tips for your project’s game architecture

2. Some perspective on where Unity falls short
Outline

1. Game loops and simulations
2. Handling interactions
3. Controls
4. Entity component system
Outline

1. Game loops and simulations
2. Handling interactions
3. Controls
4. Entity component system
Game Loop

Game loops handle all game simulation logic:

- Add / remove entities
- Handle player inputs
- Update game logic systems
- Handle movement, physics and collisions
- Generate state for the view layer
Game Loop

Game loops decouple your simulation timeline from:

- User input
- Processor speed
Unity’s Game Loop

Reset is called in the Editor when the script is attached or reset.

Start is only ever called once for a given script.

The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.

If a coroutine has yielded previously but is now due to resume their execution takes place during this part of the update.

https://docs.unity3d.com/Manual/ExecutionOrder.html
Unity’s Game Loop

Game logic in Update()?

Worry less about view interpolation

Couples simulation frequency with rendering loop
Unity’s Game Loop

Game logic in FixedUpdate()? In sync with physics loop

Easier to reason with a discrete quantized timeline

Your views will be jittery if you don’t interpolate correctly
Unity’s Game Loop

What about ordering of behavior within game loop?
Unity’s Game Loop
Unity’s Game Loop
Unity’s Game Loop

Add scripts to the custom order and drag them to reorder.

Scripts in the custom order can execute before or after the default time and are executed from top to bottom. All other scripts execute at the default time in the order they are loaded.

(Changing the order of a script may modify the meta data for more than one script.)

- WorldInput
  Default Time
  - CameraController 50
  - HUD 100
Unity’s Game Loop

You can order component priorities with a customized script execution order.

But that means you are defining one ordering for:

- Initialization
- Updates
- Teardown

Not always convenient:

- Hard to make sense of the merged ordering
- Maybe you want different orderings
You may also have cyclical dependencies between components during your initialization steps.

You could split components to solve this problem, but that may require you to update a lot of prefabs, and may increase coupling between components.

You could split initialization code between Awake() and Start(), but that’s really bad.

A good way to solve this problem is to separate your behaviors from your state, and not use Unity’s callbacks for most of your components.
Simulation

Object that encapsulates loop and state
Simulation

In Unity, there is one simulation running at all times.

You have very little control over it:

- You can’t fast forward / rewind / replay
- You can’t instantiate multiple simulations
- You can’t differentiate initialization order from update order
- You can’t define multiple initialization / update behaviors on the same component
Simulation

You can solve all of these problems by creating your own simulation class.

```java
private int ticksElapsed;
private GameState gameState;
private List<SimulationSystem> systems;

public Simulation() {
    gameState = new GameState();
    systems = new List<SimulationSystem>();
    systems.Add(new InputSystem());
    systems.Add(new AISystem());
    systems.Add(new CommandSystem());
    systems.Add(new ActionSystem());
    systems.Add(new PhysicsSystem());
    systems.Add(new DamageSystem());
    systems.Add(new EntityUpdateSystem());
    systems.Add(new ViewStateSystem());
}

public void Tick(float deltaTime) {
    foreach (SimulationSystem system in systems) {
        system.Tick(deltaTime);
    }
    ++ticksElapsed;
}
```
Multiple Simulations
Multiple Simulations

World Exploration

Level Exploration

Combat

<table>
<thead>
<tr>
<th>Action</th>
<th>Locke</th>
<th>Celes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fight</td>
<td>2063</td>
<td>1396</td>
</tr>
<tr>
<td>Morph</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Magic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terra</td>
<td>1500</td>
<td></td>
</tr>
</tbody>
</table>
Outline

1. Game loops and simulations
2. Handling interactions
3. Controls
4. Entity component system
class Comedian {
    public readonly string name;
    private bool wasSlapped;
    private Comedian facing;

    public Comedian(string name) {
        this.name = name;
    }

    public void Face(Comedian comedian) {
        facing = comedian;
    }

    public void Reset() {
        wasSlapped = false;
    }

    public void Slap() {
        wasSlapped = true;
    }

    public void Tick() {
        if (wasSlapped && (facing != null)) {
            facing.Slap();
        }
    }
}
class Comedian {
    public readonly string name;
    private bool wasSlapped;
    private Comedian facing;

    public Comedian(string name) {
        this.name = name;
    }

    public void Face(Comedian comedian) {
        facing = comedian;
    }

    public void Reset() {
        wasSlapped = false;
    }

    public void Slap() {
        wasSlapped = true;
    }

    public void Tick() {
        if ((wasSlapped) && (facing != null)) {
            facing.Slap();
        }
    }
}
class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[comedianIndex];
            var nextComedian = comedians[(comedianIndex + 1) % comedians.Count];
            comedian.Face(nextComedian);
        }

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}
Interaction Resolution

class Stage {
  private List<Comedian> comedians;

  public Stage() {
    comedians = new List<Comedian>();
    comedians.Add(new Comedian("Moe"));
    comedians.Add(new Comedian("Curly"));
    comedians.Add(new Comedian("Larry"));

    for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
      var comedian = comedians[comedianIndex];
      var nextComedian = comedians[(comedianIndex + 1) % comedians.Count];
      comedian.Face(nextComedian);
    }
    comedians[0].Slap();
  }

  public void Tick() {
    foreach (var comedian in comedians) {
      comedian.Tick();
    }
  }
}

Slap!
class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianInde
        var comedian = comedians[comedianIndex];
        var nextComedian = comedians[(comedianIndex + 1) % comedians.Count];
        comedian.Face(nextComedian);

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
        }
        comedian.Reset();
    }
}
Interaction Resolution

class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[comedianIndex];
            var nextComedian = comedians[(comedianIndex + 1) % comedians.Count];
            comedian.Face(nextComedian);
        }

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
        }
        comedian.Reset();
    }
}
Interaction Resolution

class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[comedianIndex];
            var nextComedian = comedians[(comedianIndex + 1) % comedians.Count];
            comedian.Face(nextComedian);
        }

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}
Interaction Resolution

class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[(comedianIndex + 1) % comedians.Count];
            var nextComedian = comedians[comedianIndex];
            comedian.Face(nextComedian);
        }
        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}
Interaction Resolution

class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[(comedianIndex + 1) % comedians.Count];
            var nextComedian = comedians[comedianIndex];
            comedian.Face(nextComedian);
        }
        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}

Slap!
class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[(comedianIndex + 1) % comedians.Count];
            var nextComedian = comedians[comedianIndex];
            comedian.Face(nextComedian);
        }

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}
Interaction Resolution

class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[(comedianIndex + 1) % comedians.Count];
            var nextComedian = comedians[comedianIndex];
            comedian.Face(nextComedian);
        }

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}
Interaction Resolution

class Stage {
  private List<Comedian> comedians;

  public Stage() {
    comedians = new List<Comedian>();
    comedians.Add(new Comedian("Moe"));
    comedians.Add(new Comedian("Curly");
    comedians.Add(new Comedian("Larry");

    for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
      var comedian = comedians[comedianIndex + 1] % comedians.Count;
      var nextComedian = comedians[comedianIndex];
      comedian.Face(nextComedian);
    }
    comedians[0].Slap();
  }

  public void Tick() {
    foreach (var comedian in comedians) {
      comedian.Tick();
      comedian.Reset();
    }
  }
}
class Stage {
    private List<Comedian> comedians;

    public Stage() {
        comedians = new List<Comedian>();
        comedians.Add(new Comedian("Moe"));
        comedians.Add(new Comedian("Curly"));
        comedians.Add(new Comedian("Larry"));

        for (int comedianIndex = 0; comedianIndex < comedians.Count; ++comedianIndex) {
            var comedian = comedians[(comedianIndex + 1) % comedians.Count];
            var nextComedian = comedians[comedianIndex];
            comedian.Face(nextComedian);
        }

        comedians[0].Slap();
    }

    public void Tick() {
        foreach (var comedian in comedians) {
            comedian.Tick();
            comedian.Reset();
        }
    }
}
Interaction Resolution

Do we care? Maybe.

- Turn based simulations
- Competitive games
- Consistently biased small advantages can add up
Double buffering

Game Loop

Read
Old Game State

Write
New Game State
Double buffering
Double buffering

After swapping, instead of a buffer clear on the active buffer, we do a buffer copy.

The buffer copy can be unnecessarily expensive:

- Game states can be large
- Usually only a small portion of the game state changes each tick

Potentially use this concept on pieces of state, or avoid it completely.
Message queues

- Initiate slap
- Propagate slaps
Message queues
Message queues

1. Initiate slap
2. Propagate slaps
3. Perform slap
Outline

1. Game loops and simulations
2. Handling interactions
3. Controls
4. Entity component system
private void Update() {
    // Press up arrow key => attempt to jump
    if (Input.GetKeyDown(KeyCode.UpArrow)) {
        Jump();
    }

    // Press space bar => fire lemons out of your gun
    if (Input.GetKeyDown(KeyCode.Space)) {
        FireLemon();
    }
}
Controls - “Command” Pattern

```java
public class JumpCommand : Command {
    public override void Execute() {
        // Jump
    }
}

public class FireLemonCommand : Command {
    public override void Execute() {
        // Fire a lemon.
    }
}

Dictionary<KeyCode, Command> commandsByKeyCode;

private void InitializeCommands() {
    commandsByKeyCode(KeyCode.UpArrow) = new JumpCommand();
    commandsByKeyCode(KeyCode.Space) = new FireLemonCommand();
}

private void Update()
```

```java
    foreach (KeyCode keyCode in commandsByKeyCode.Keys) {
        if (Input.GetKeyDown(keyCode)) {
            Command command = commandsByKeyCode[keyCode];
            command.Execute();
        }
    }
```
Controls - “Command” Pattern

```csharp
public class JumpCommand : Command {
    public override void Execute() {
        // Jump
    }
}

public class FireLemonCommand : Command {
    public override void Execute() {
        // Fire a lemon.
    }
}

Dictionary<KeyCode, Command> commandsByKeyCode;

private void InitializeCommands() {
    commandsByKeyCode[KeyCode.UpArrow] = new JumpCommand();
    commandsByKeyCode[KeyCode.Space] = new FireLemonCommand();
}

private void Update() {
    foreach (KeyCode keyCode in commandsByKeyCode.Keys) {
        if (Input.GetKeyDown(keyCode)) {
            Command command = commandsByKeyCode[keyCode];
            command.Execute();
        }
    }
}
```

Controls - "Command" Pattern

```csharp
public class JumpCommand : Command {
    public override void Execute() {
        // Jump
    }
}

public class FireLemonCommand : Command {
    public override void Execute() {
        // Fire a lemon.
    }
}

Dictionary<KeyCode, Command> commandsByKeyCode;

private void InitializeCommands() {
    commandsByKeyCode[KeyCode.UpArrow] = new JumpCommand();
    commandsByKeyCode[KeyCode.Space] = new FireLemonCommand();
}

private void Update() {
    foreach (KeyCode keyCode in commandsByKeyCode.Keys) {
        if (Input.GetKeyDown(keyCode)) {
            Command command = commandsByKeyCode[keyCode];
            command.Execute();
        }
    }
}
```
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command command;

    public InputHandler(string name, KeyCode keyCode, Command command) {
        this.name = name;
        this.keyCode = keyCode;
        this.command = command;
    }

    public void HandleInput() {
        if (Input.GetKeyDown(keyCode)) {
            command.Execute();
        }
    }
}
Controls - Input Handlers

```csharp
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command command;

    public InputHandler(string name, KeyCode keyCode, Command command) {
        this.name = name;
        this.keyCode = keyCode;
        this.command = command;
    }

    public void HandleInput() {
        if (Input.GetKeyDown(keyCode)) {
            command.Execute();
        }
    }
}

private List<InputHandler> inputHandlers;

private void InitializeInputHandlers() {
    inputHandlers.Add(new InputHandler("Jump", KeyCode.UpArrow, new JumpCommand()));
    inputHandlers.Add(new InputHandler("Fire", KeyCode.Space, new FireLemonCommand()));
}
```
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command command;

    public InputHandler(string name, KeyCode keyCode, Command command) {
        this.name = name;
        this.keyCode = keyCode;
        this.command = command;
    }

    public void HandleInput() {
        if (Input.GetKeyDown(keyCode)) {
            command.Execute();
        }
    }
}

private List<InputHandler> inputHandlers;

private void InitializeInputHandlers() {
    inputHandlers.Add(new InputHandler("Jump", KeyCode.UpArrow, new JumpCommand()));
    inputHandlers.Add(new InputHandler("Fire", KeyCode.Space, new FireLemonCommand()));
}

private void Update() {
    foreach (var inputHandler in inputHandlers) {
        inputHandler.HandleInput();
    }
}
Controls - Input with duration

Some controls are kept activated.

For example, holding a button down for movement.
Controls - Input with duration

Could try this:

- Activation command on key down
- Deactivation command on key up
Controls - Input with duration

Could try this:

- Activation command on key down
- Deactivation command on key up

Couple problems:

- Callbacks may be missed
- Conflicting controls could be a problem
Controls - Input with duration

You could poll the input state, and keep track of input handler activation.
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command activateCommand;
    public Command deactivateCommand;

    private bool isActivated;

    public InputHandler(string name, KeyCode keyCode, Command activateCommand, Command deactivateCommand) {
        this.name = name;
        this.keyCode = keyCode;
        this.activateCommand = activateCommand;
        this.deactivateCommand = deactivateCommand;
    }

    public void HandleInput() {
        if (isActivated := Input.GetKey(keyCode)) {
            if (Input.GetKey(keyCode)) {
                Activate();
            } else {
                Deactivate();
            }
        }
    }

    private void Activate() {
        isActivated = true;
        if (activateCommand != null) {
            activateCommand.Execute();
        }
    }

    private void Deactivate() {
        isActivated = false;
        if (deactivateCommand != null) {
            deactivateCommand.Execute();
        }
    }
}
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command activateCommand;
    public Command deactivateCommand;

    private bool isActivated;

    public InputHandler(string name, KeyCode keyCode, Command activateCommand, Command deactivateCommand) {
        this.name = name;
        this.keyCode = keyCode;
        this.activateCommand = activateCommand;
        this.deactivateCommand = deactivateCommand;
    }

    public void HandleInput() {
        if (isActivated != Input.GetKey(keyCode)) {
            if (Input.GetKey(keyCode)) {
                Activate();
            } else {
                Deactivate();
            }
        }
    }

    private void Activate() {
        isActivated = true;
        if (activateCommand != null) {
            activateCommand.Execute();
        }
    }

    private void Deactivate() {
        isActivated = false;
        if (deactivateCommand != null) {
            deactivateCommand.Execute();
        }
    }
}
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command activateCommand;
    public Command deactivateCommand;

    private bool isActivated;

    public InputHandler(string name, KeyCode keyCode, Command activateCommand, Command deactivateCommand) {
        this.name = name;
        this.keyCode = keyCode;
        this.activateCommand = activateCommand;
        this.deactivateCommand = deactivateCommand;
    }

    public void HandleInput() {
        if (isActivated != Input.GetKey(keyCode)) {
            if (Input.GetKey(keyCode)) {
                Activate();
            } else {
                Deactivate();
            }
        }
    }

    private void Activate() {
        isActivated = true;
        if (activateCommand != null) {
            activateCommand.Execute();
        }
    }

    private void Deactivate() {
        isActivated = false;
        if (deactivateCommand != null) {
            deactivateCommand.Execute();
        }
    }
}
public class InputHandler {
    public readonly string name;
    public KeyCode keyCode;
    public Command activateCommand;
    public Command deactivateCommand;
    private bool isActivated;

    public InputHandler(string name, KeyCode keyCode, Command activateCommand, Command deactivateCommand) {
        this.name = name;
        this.keyCode = keyCode;
        this.activateCommand = activateCommand;
        this.deactivateCommand = deactivateCommand;
    }

    public void HandleInput() {
        if (isActivated != Input.GetKey(keyCode)) {
            if (Input.GetKey(keyCode)) {
                Activate();
            } else {
                Deactivate();
            }
        }
    }

    private void Activate() {
        isActivated = true;
        if (activateCommand != null) {
            activateCommand.Execute();
        }
    }

    private void Deactivate() {
        isActivated = false;
        if (deactivateCommand != null) {
            deactivateCommand.Execute();
        }
    }
}
Controls - Input with duration

Let’s update our input handlers for “Jump” and “Fire”.

```csharp
private void InitializeInputHandlers() {
    inputHandlers.Add(new InputHandler("Jump", KeyCode.UpArrow, new JumpCommand(), null));
    inputHandlers.Add(new InputHandler("Fire", KeyCode.Space, new FireLemonCommand(), null));
}
```
Controls - Input with duration

Now add input handlers for moving left and right.

```csharp
private void InitializeInputHandlers() {
    inputHandlers.Add(new InputHandler("Move Left", KeyCode.LeftArrow, new BeginMoveCommand(new Vector3f(-1, 0, 0)), new EndMoveCommand()));
    inputHandlers.Add(new InputHandler("Move Right", KeyCode.RightArrow, new BeginMoveCommand(new Vector3f(1, 0, 0)), new EndMoveCommand()));
    inputHandlers.Add(new InputHandler("Jump", KeyCode.UpArrow, new JumpCommand(), null));
    inputHandlers.Add(new InputHandler("Fire", KeyCode.Space, new FireLemonCommand(), null));
}
```
Controls - Simultaneous inputs

What about conflicting controls?

What happens when you press multiple buttons at the same time?
Controls - Simultaneous inputs
Controls - Simultaneous inputs
Controls - Simultaneous inputs
Controls - Simultaneous inputs
Controls - Simultaneous inputs
Controls - Simultaneous inputs
Controls - Simultaneous inputs

Sometimes you want to handle only one input at a time out of a set of inputs.
Controls - Simultaneous inputs

Sometimes you want to handle only one input at a time out of a set of inputs.

Priority of input may be based on various factors:

- How recent is the input?
- What are the gameplay effects of the input?
Controls - Simultaneous inputs

Sometimes you want to handle only one input at a time out of a set of inputs.

Priority of input may be based on various factors:

- How recent is the input?
- What are the gameplay effects of the input?

Are these relationships gameplay rules, or input rules?
Controls - Simultaneous inputs

You can use state machines, not only for animation, but also for gameplay logic.

In our movement system, you are either idle, moving left, or moving right.
Controls - Simultaneous inputs
Controls - Simultaneous inputs

Let’s examine our movement code again.

Releasing one direction button could end movement in the opposite direction.

We don’t want that.

```csharp
private void InitializeInputHandlers() {
    inputHandlers.Add(new InputHandler("Move Left", KeyCode.LeftArrow, new BeginMoveCommand(new Vector3f(-1, 0, 0)), new EndMoveCommand()));
    inputHandlers.Add(new InputHandler("Move Right", KeyCode.RightArrow, new BeginMoveCommand(new Vector3f(1, 0, 0)), new EndMoveCommand()));
    inputHandlers.Add(new InputHandler("Jump", KeyCode.UpArrow, new JumpCommand(), null));
    inputHandlers.Add(new InputHandler("Fire", KeyCode.Space, new FireLemonCommand(), null));
}
```
Controls - Simultaneous inputs

Instead of requesting velocity modifications, we request state transitions.

```csharp
private void InitializeInputHandlers() {
    inputHandlers.Add(new InputHandler("Move Left", KeyCode.LeftArrow, new BeginMoveLeftCommand(), new EndMoveLeftCommand()));
    inputHandlers.Add(new InputHandler("Move Right", KeyCode.RightArrow, new BeginMoveRightCommand(), new EndMoveRightCommand()));
    inputHandlers.Add(new InputHandler("Jump", KeyCode.UpArrow, new JumpCommand(), null));
    inputHandlers.Add(new InputHandler("Fire", KeyCode.Space, new FireLemonCommand(), null));
}
```
Outline

1. Game loops and simulations
2. Handling interactions
3. Controls
4. Entity component system
Entity Component System (ECS)

All interactive objects in your game are Entities.

Each Entity is associated with a collection of Components.

Plural quantities of certain types of Components may exist on an Entity.
Entity Component System (ECS)
Entity Component System (ECS)
Entity Component System (ECS)

- Entity
  - Transform Component
  - Collision Component
  - Obstacle Component
  - Movement Component
  - View Component

- Entity
  - Transform Component
  - Collision Component
  - Destructible Component
  - Reward Component
  - View Component

- Score Component

- Scoreboard:
  - Home: 3, Visitor: 2
  - Period: 2
  - Player 1: 24, Penalties: 1:14, Player 2: 34, Penalties: 1:56
  - Period: 2
  - Fair-Play Time:
    - Player: 15, Time: 2:32
    - Player: 20, Time: 2:27
Unity’s ECS
Unity’s ECS
Unity’s ECS
Unity’s ECS

Two problems with Unity’s ECS:

- No view / model separation
  - Probably not a huge problem for your final projects
  - Becomes a problem when you make a networked multiplayer game
Two problems with Unity’s ECS:

- No view / model separation
  - Probably not a huge problem for your final projects
  - Becomes a problem when you make a networked multiplayer game

- No behavior / state separation
  - You can build this on top of Unity’s system, by creating your own simulation code
Behavior and State Separation

It may help to separate behaviors from state.

Why?

- Behaviors need to be organized and ordered in the game loop, but state does not.
- Modularization of behaviors is not clear if they are in the same class as the state.
Behavior and State Separation

System 1  |  Component A
System 2  |  Component B
System 3  |  Component C
System 4  |  Component D
          |  Component E
          |  Component F
          |  Component G
Behavior and State Separation

System 1
System 2
System 3
System 4

Component A
Component B
Component C
Component D
Component E
Component F
Component G
Behavior and State Separation

System 1  |  Component A
System 2  |  Component B
System 3  |  Component C
System 4  |  Component D
          |  Component E
          |  Component F
          |  Component G
Behavior and State Separation

- System 1
- System 2
- System 3
- System 4

- Component A
- Component B
- Component C
- Component D
- Component E
- Component F
- Component G
Behavior and State Separation

- System 1
- System 2
- System 3
- System 4

- Component A
- Component B
- Component C
- Component D
- Component E
- Component F
- Component G
Review
Review

1. Game loops and simulations
   a. Unity calls your code via callbacks, such as Awake(), Start(), FixedUpdate(), Update()
   b. You can customize the script execution order (be careful)
   c. Consider making a simulation class / component, to gain control
Review

1. **Game loops and simulations**
   a. Unity calls your code via callbacks, such as Awake(), Start(), FixedUpdate(), Update()
   b. You can customize the script execution order (be careful)
   c. Consider making a simulation class / component, to gain control

2. **Handling interactions**
   a. Double buffering
   b. Message queues
Review

1. Game loops and simulations
   a. Unity calls your code via callbacks, such as Awake(), Start(), FixedUpdate(), Update()
   b. You can customize the script execution order (be careful)
   c. Consider making a simulation class / component, to gain control

2. Handling interactions
   a. Double buffering
   b. Message queues

3. Controls
   a. Commands and input handlers
   b. Resolve conflicts in your gameplay logic, possibly by using state machines
Review

1. Game loops and simulations
   a. Unity calls your code via callbacks, such as Awake(), Start(), FixedUpdate(), Update()
   b. You can customize the script execution order (be careful)
   c. Consider making a simulation class / component, to gain control

2. Handling interactions
   a. Double buffering
   b. Message queues

3. Controls
   a. Commands and input handlers
   b. Resolve conflicts in your gameplay logic, possibly by using state machines

4. Entity component system
   a. Unity’s ECS is very versatile
   b. You can write your own classes / components to extend it
      i. View / model separation
      ii. Behavior / state separation
Contact

Happy to provide help on final projects and career planning!

Reach out to me at mchung@pocketgems.com