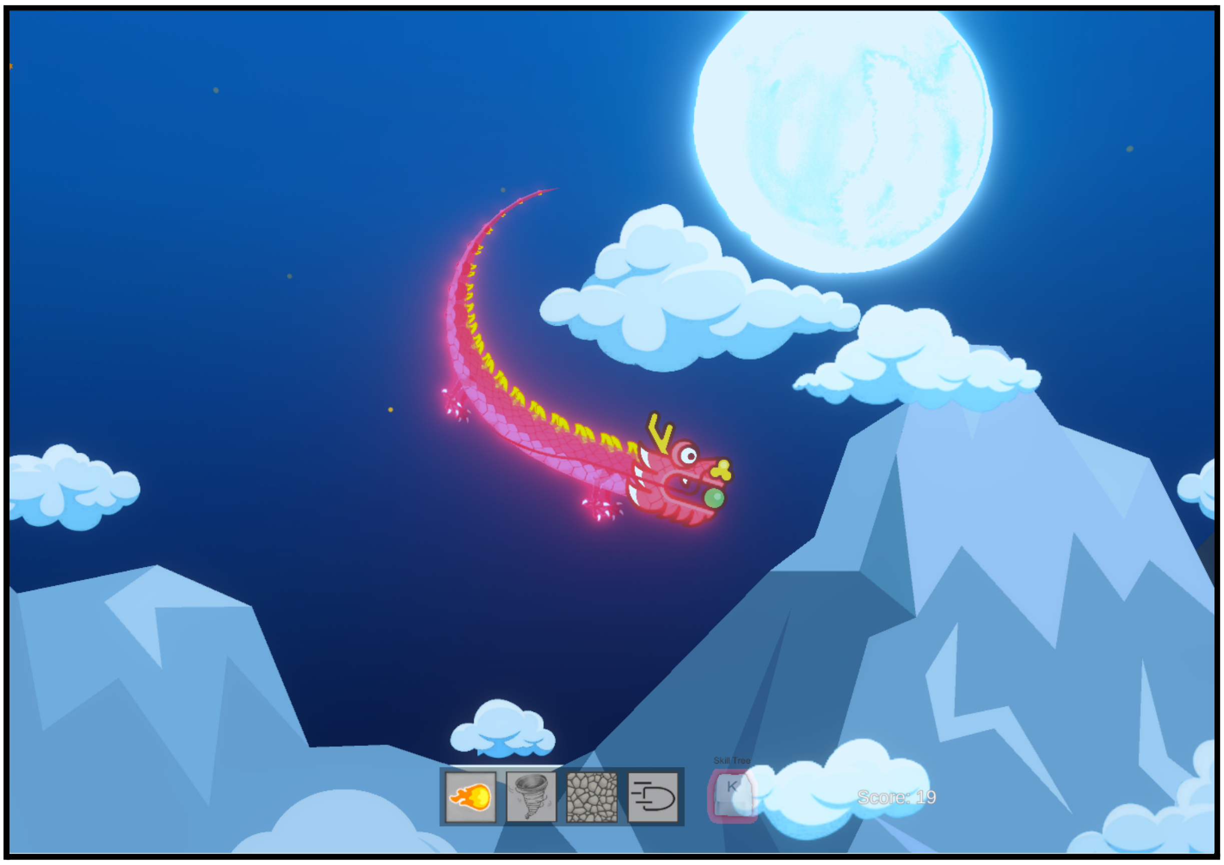
DragonSnake



This project is a continuation/remake on the last assignment "*Snaked list*" where I created a game where you play as a chinese dragon with 4 abilities:

1. **Fireball** - when colliding with enemy head, turns whole snake into food
2. **Stonestate** - protects against other abilities for 0.5s
3. **Fireball** - when colliding with enemy head, turns whole snake into food
4. **Whirlwind** - cuts other snakes when colliding, turning half the snake into food.
5. **Dash** - short burst of speed

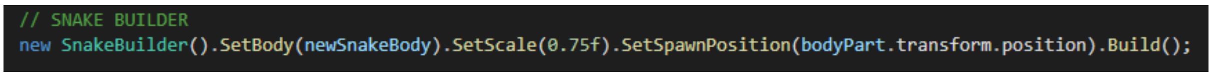
At its current state, you get to face an AI able to attack you and steal your food.

Warning: there might be some "zombie" code and scripts left behind from the last assignment. But I've tried to include screenshots here of the most important parts.

Design patterns

Builder - *in SnakeBuilder.cs*

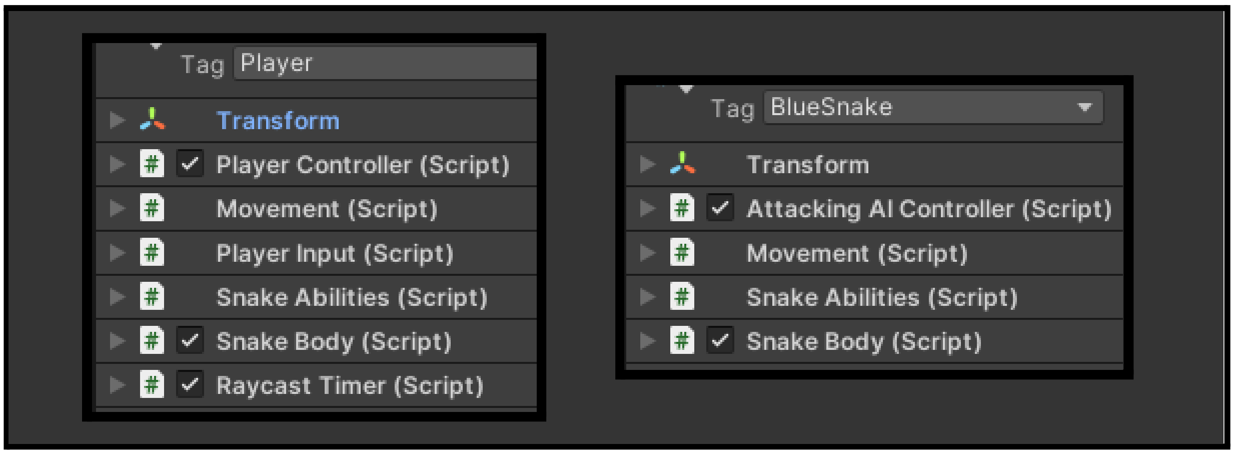
I used the builder pattern to create new snakes, for example when a snake splits into two by a bomb. I implemented it with method chaining, by having each method inside the builder return itself ("this").



```
// SNAKE BUILDER
new SnakeBuilder().SetBody(newSnakeBody).SetScale(0.75f).SetSpawnPosition(bodyPart.transform.position).Build();
```

Component - *in PlayerController.cs, AttackingAISnakeController.cs etc.*

I used the component pattern to easily be able to make different types of snakes that share common components but also have unique ones. All snakes have a movement component, but while the movement of the player is determined by input, the movement of an AI snake is determined by a state machine.



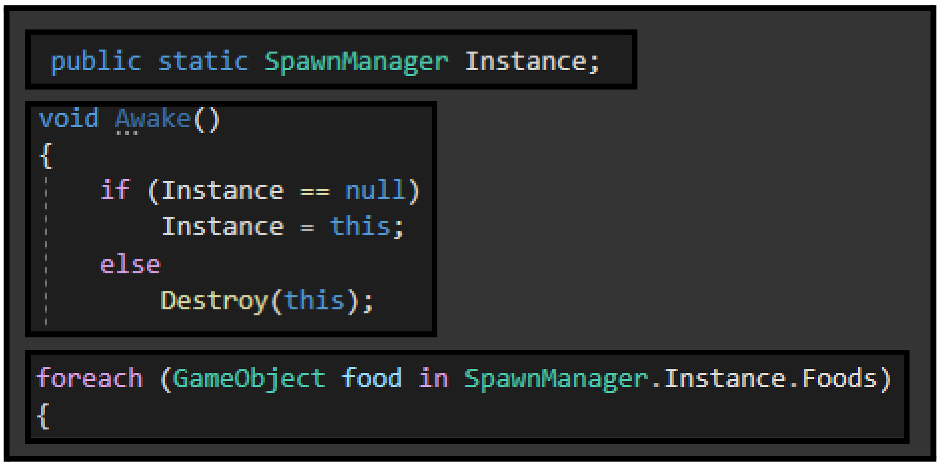
```
Tag  Player
Transform
✓ Player Controller (Script)
Movement (Script)
Player Input (Script)
Snake Abilities (Script)
✓ Snake Body (Script)
✓ Raycast Timer (Script)
```

```
Tag  BlueSnake
Transform
✓ Attacking AI Controller (Script)
Movement (Script)
Snake Abilities (Script)
✓ Snake Body (Script)
```

Another example is found at the end of this paper, demonstrating the usage of components

Singleton - *SpawnManager.cs*

I made the spawn manager a singleton, to be able to get global access to all the active food/bombs.



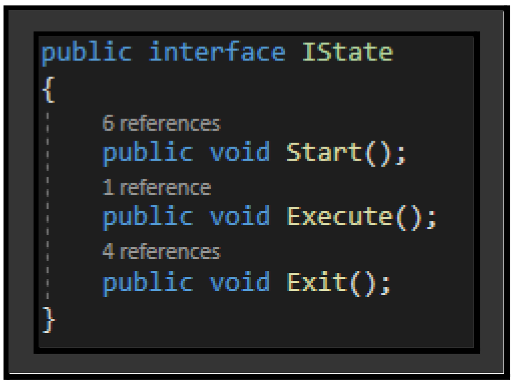
```
public static SpawnManager Instance;
```

```
void Awake()
{
    if (Instance == null)
        Instance = this;
    else
        Destroy(this);
```

```
foreach (GameObject food in SpawnManager.Instance.Foods)
{
```

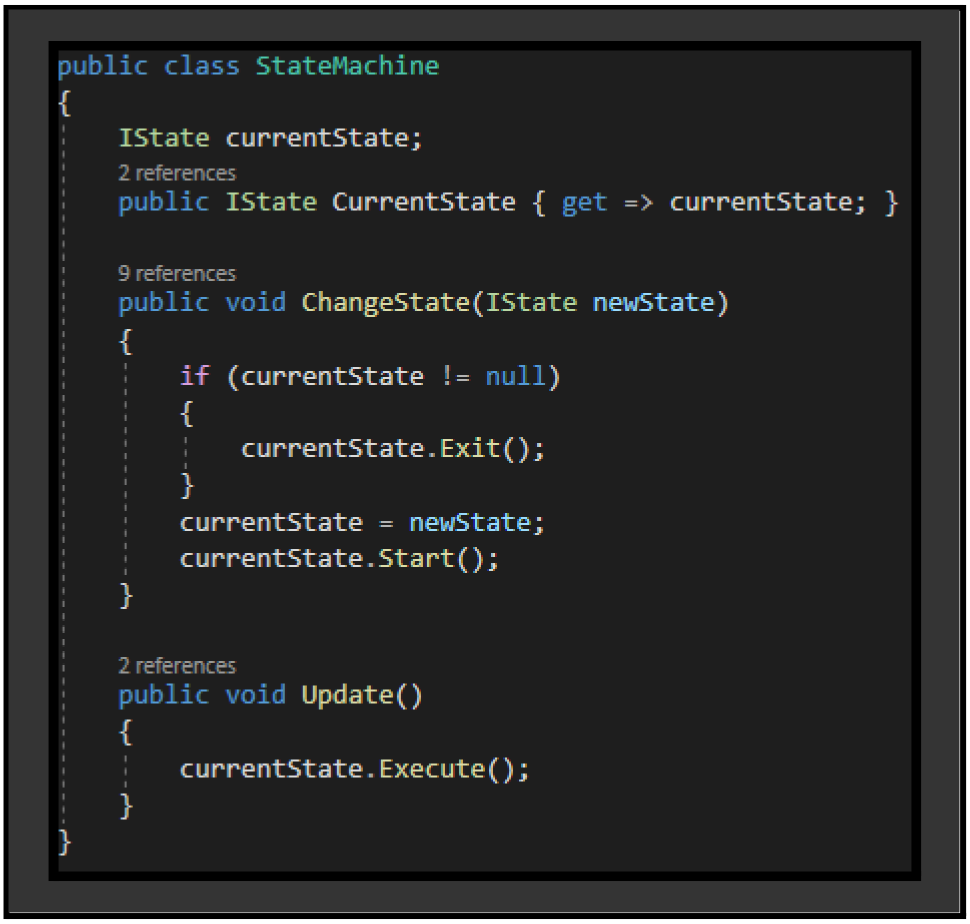
Finite-State machine - *StateMachine.cs, IState.cs, MoveTowardFood.cs etc.*

In my implementation of a state machine, instead of making Enums for states and events, I created a new class for each state, all implementing the interface IState.



```
public interface IState
{
    6 references
    public void Start();
    1 reference
    public void Execute();
    4 references
    public void Exit();
}
```

The state machine can change state and update/execute the current state like shown below.

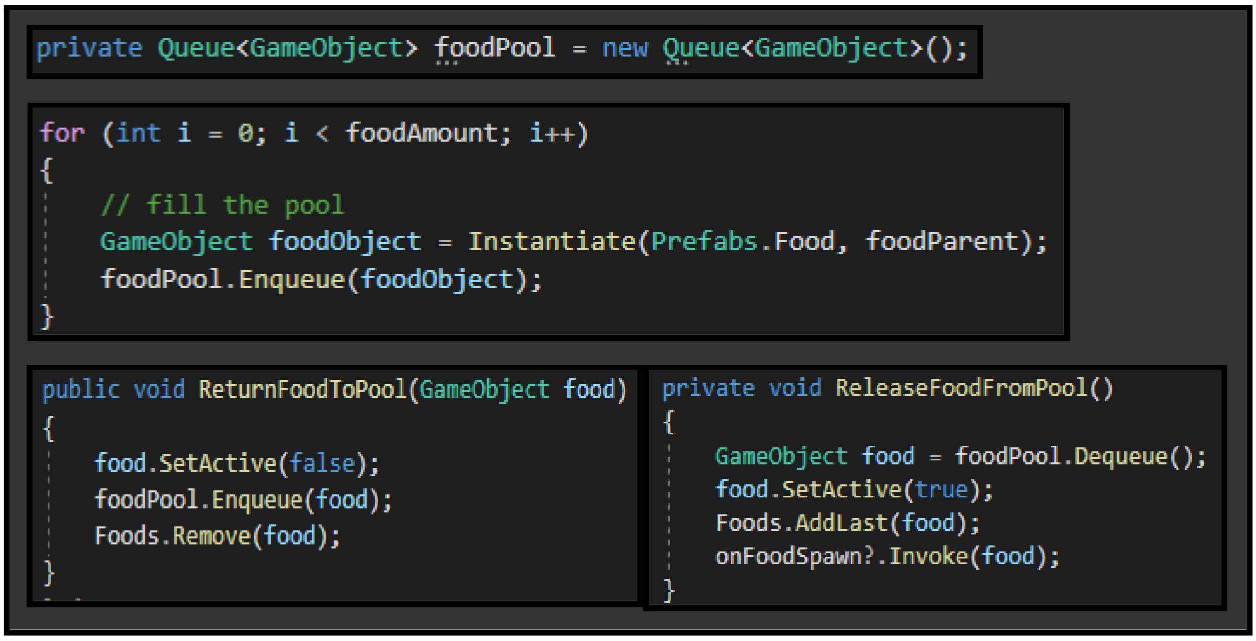


```
public class StateMachine
{
    IState currentState;
    2 references
    public IState CurrentState { get => currentState; }

    9 references
    public void ChangeState(IState newState)
    {
        if (currentState != null)
        {
            currentState.Exit();
        }
        currentState = newState;
        currentState.Start();
    }

    2 references
    public void Update()
    {
        currentState.Execute();
    }
}
```

So far, I have added 3 different states, **MoveTowardFoodState**, **LongRangeAttackState** and **ShortRangeAttackState**. They are all implemented in **AttackingAIController**.
More examples of the state machine are shown at the end of this paper.

Object Pool - *in SpawnManager.cs*

I implemented an object pool for the spawning/despawning of food (only for the sake of the assignment). The screenshots below demonstrate how it was implemented.



```
private Queue<GameObject> foodPool = new Queue<GameObject>();
```

```
for (int i = 0; i < foodAmount; i++)
{
    // fill the pool
    GameObject foodObject = Instantiate(Prefabs.Food, foodParent);
    foodPool.Enqueue(foodObject);
}
```

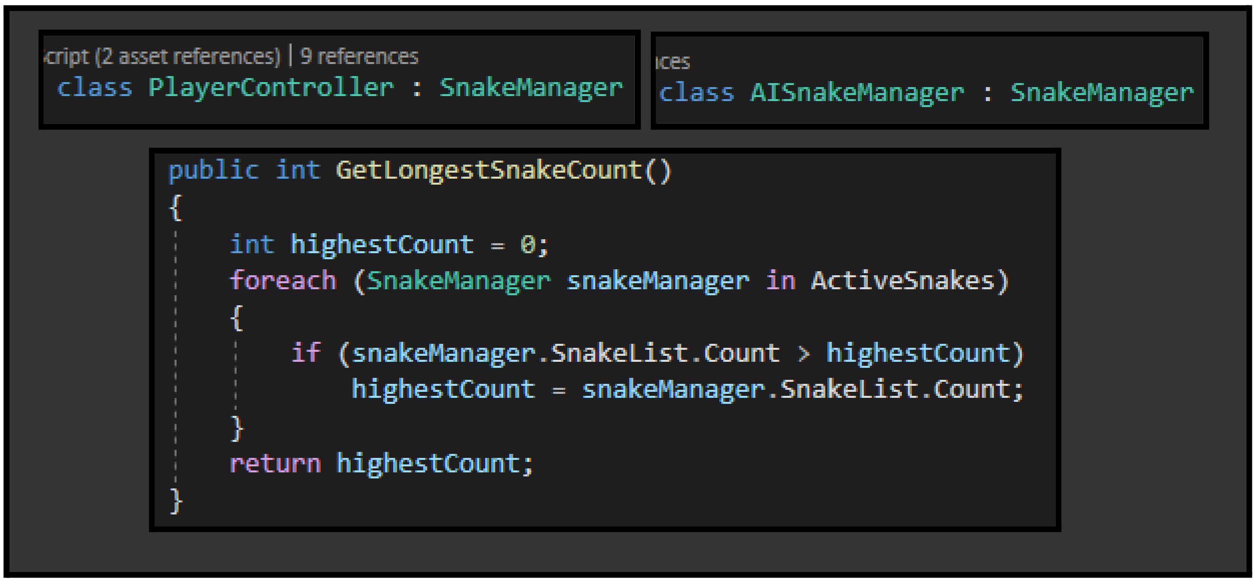
```
public void ReturnFoodToPool(GameObject food)
{
    food.SetActive(false);
    foodPool.Enqueue(food);
    Foods.Remove(food);
}
```

```
private void ReleaseFoodFromPool()
{
    GameObject food = foodPool.Dequeue();
    food.SetActive(true);
    Foods.AddLast(food);
    onFoodSpawn?.Invoke(food);
}
```

Shows the different parts involved with the object pool

Composite - *in SnakeManager.cs, PlayerController.cs etc.*

I used the composite pattern to get the value of variables that different classes had in common. For example, all types of snakes (**PlayerSnake**, **AttackingAISnake**, **NonAttackingAISnake**) have a Linked List of game objects (which represents the body of the snake). And all snakes inherit from the abstract class **SnakeManager**.



```
cript (2 asset references) | 9 references
class PlayerController : SnakeManager
```

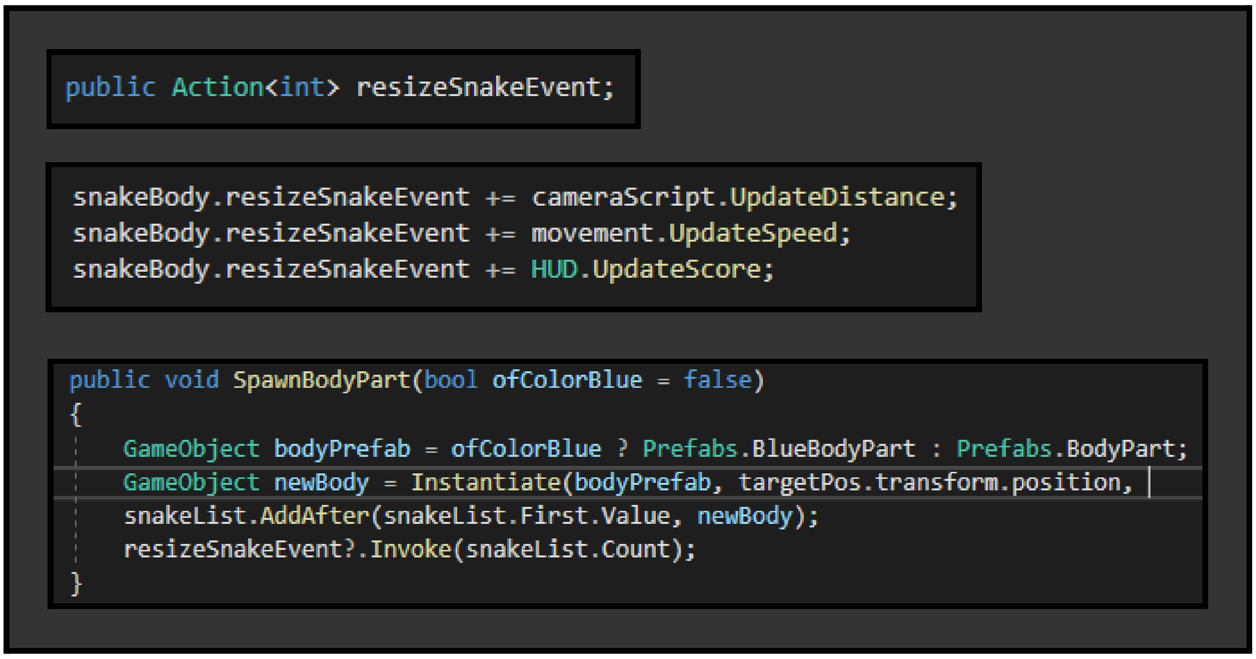
```
ices
class AISnakeManager : SnakeManager
```

```
public int GetLongestSnakeCount()
{
    int highestCount = 0;
    foreach (SnakeManager snakeManager in ActiveSnakes)
    {
        if (snakeManager.SnakeList.Count > highestCount)
            highestCount = snakeManager.SnakeList.Count;
    }
    return highestCount;
}
```

Simplified illustration of how the pattern could be used

Observer pattern

I’m not sure what actually counts as an observer pattern, for example if you actually need a list of subscribers. I have implemented simple events. One example is the event “**resizeSnakeEvent**”, which is invoked whenever the size of the snake changes (eats or splits), with “snakeList.Count” being passed as argument.



```
public Action<int> resizeSnakeEvent;
```

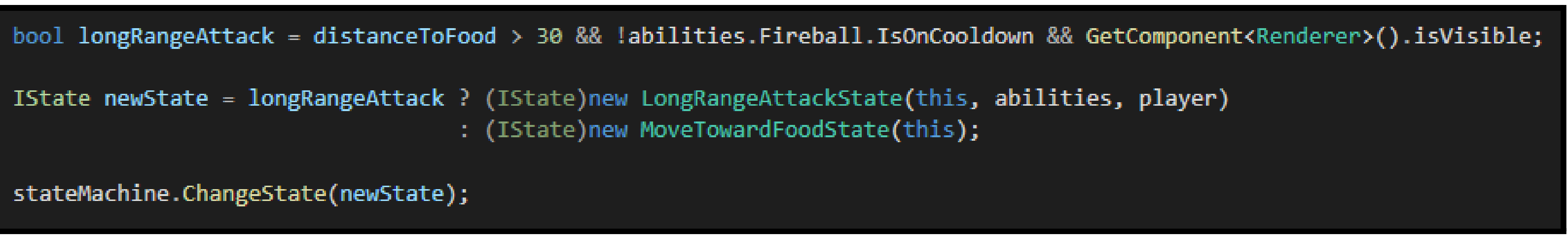
```
snakeBody.resizeSnakeEvent += cameraScript.UpdateDistance;
snakeBody.resizeSnakeEvent += movement.UpdateSpeed;
snakeBody.resizeSnakeEvent += HUD.UpdateScore;
```

```
public void SpawnBodyPart(bool ofColorBlue = false)
{
    GameObject bodyPrefab = ofColorBlue ? Prefabs.BlueBodyPart : Prefabs.BodyPart;
    GameObject newBody = Instantiate(bodyPrefab, targetPos.transform.position,
    snakeList.AddAfter(snakeList.First.Value, newBody);
    resizeSnakeEvent?.Invoke(snakeList.Count);
}
```

Illustration of how events where implemented

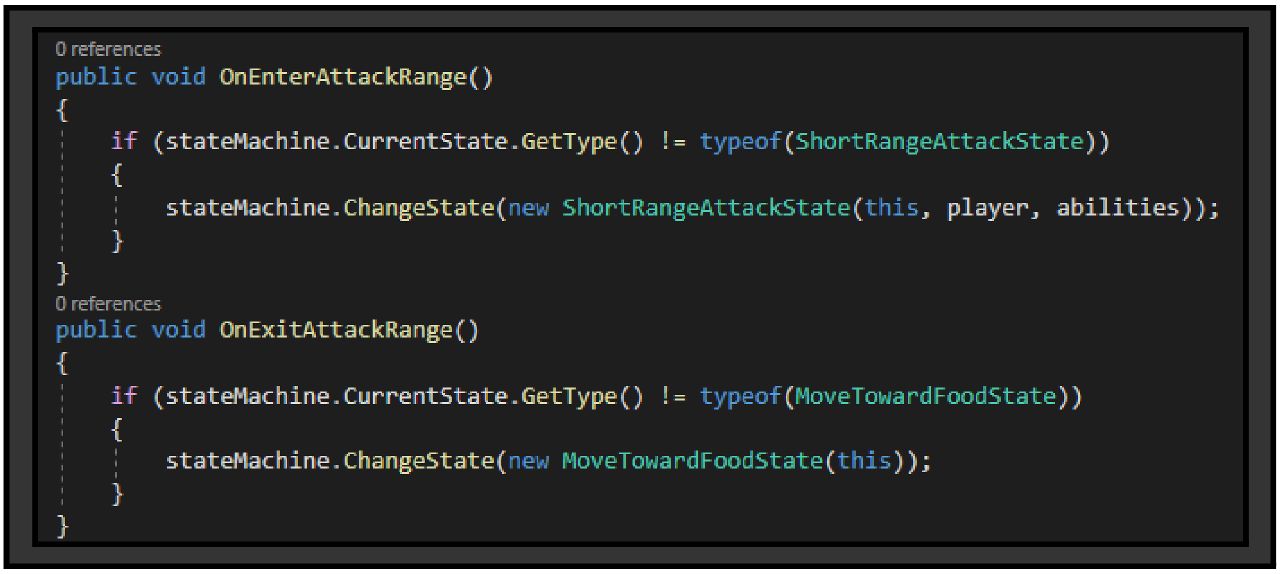
More examples

State machine



```csharp
bool longRangeAttack = distanceToFood > 30 && !abilities.Fireball.IsOnCooldown && GetComponent<Renderer>().isVisible;

IState newState = longRangeAttack ? (IState)new LongRangeAttackState(this, abilities, player)
                                  : (IState)new MoveTowardFoodState(this);

stateMachine.ChangeState(newState);
```

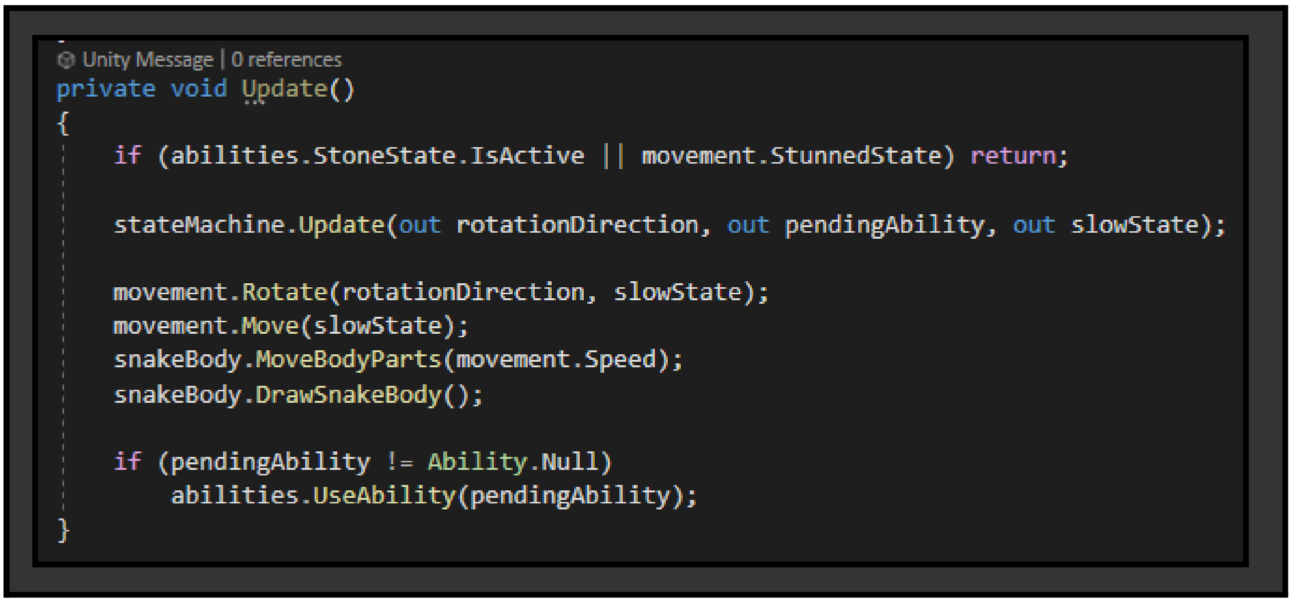
Illustration of how a new state is decided after eating food



```csharp
0 references
public void OnEnterAttackRange()
{
    if (stateMachine.CurrentState.GetType() != typeof(ShortRangeAttackState))
    {
        stateMachine.ChangeState(new ShortRangeAttackState(this, player, abilities));
    }
}
0 references
public void OnExitAttackRange()
{
    if (stateMachine.CurrentState.GetType() != typeof(MoveTowardFoodState))
    {
        stateMachine.ChangeState(new MoveTowardFoodState(this));
    }
}
```

Illustration of how the states are changed when entering/exiting attack range

Components



```csharp
Unity Message | 0 references
private void Update()
{
    if (abilities.StoneState.IsActive || movement.StunnedState) return;

    stateMachine.Update(out rotationDirection, out pendingAbility, out slowState);

    movement.Rotate(rotationDirection, slowState);
    movement.Move(slowState);
    snakeBody.MoveBodyParts(movement.Speed);
    snakeBody.DrawSnakeBody();

    if (pendingAbility != Ability.Null)
        abilities.UseAbility(pendingAbility);
}
```