Приложение 1

PlayerController.cs

```
using UnityEngine;


[RequireComponent(typeof(CharacterController))]
[RequireComponent(typeof(InputSystem))]
public class PlayerController : MonoBehaviour
{
    [Header("Настройки персонажа")] [SerializeField]
    private float _speed = 1.0f;

    [SerializeField] private float _sprintSpeed = 1.1f;
    [SerializeField] private float _accelerationSpeed = 2.0f;
    [SerializeField] private float _turnSpeed;

    private float speed;
    private float _animationBlend;

    public int _animationIDSpeed;
    private bool _hasAnimator;

    private InputSystem _input;
    private CharacterController _characterController;
    private Animator _animator;

    public bool IsLive { get; set; } = true;

    void Start()
    {
        _hasAnimator = TryGetComponent(out _animator);

        _input = GetComponent<InputSystem>();
        _characterController = GetComponent<CharacterController>();
        _animator = GetComponent<Animator>();
    AssignAnimationIDs();
}

void Update()
{
    _hasAnimator = TryGetComponent(out _animator);

    Move();
    Turn();
}

private void AssignAnimationIDs()
{
    _animationIDSpeed = Animator.StringToHash("Speed_f");
}

private void Move()
{
    if (IsLive)
    {
        var targetSpeed = _input.Sprint ? _sprintSpeed : _speed;

        if (_input.Move == Vector2.zero)
        {
            targetSpeed = 0;
        }

        var currentSpeed =
            new Vector2(_characterController.velocity.x, _characterController.velocity.y).magnitude;

        if (currentSpeed > targetSpeed || currentSpeed > targetSpeed)
        {
            speed = Mathf.Lerp(currentSpeed, targetSpeed, Time.deltaTime * _accelerationSpeed);
```

```
                speed = Mathf.Round(_speed / 1000) * 1000;
            }
            else
            {
                speed = targetSpeed;
            }

            Vector3 inputDirection = new Vector3(_input.Move.x, 0, _input.Move.y);

            _characterController.Move(inputDirection * speed * Time.deltaTime);

            _animationBlend = Mathf.Lerp(_animationBlend, targetSpeed, Time.deltaTime * _accelerationSpeed);

            _animator.SetFloat(_animationIDSpeed, _animationBlend);
        }
    }

    private void Turn()
    {
        if (IsLive)
        {
            if (_input.Move != Vector2.zero)
            {
                Quaternion toTurn = Quaternion.LookRotation(new Vector3(_input.Move.x, 0, _input.Move.y), 
Vector3.up);

                transform.rotation = Quaternion.RotateTowards(transform.rotation, toTurn, _turnSpeed * 
Time.deltaTime);
            }
        }
    }
}
```

Inventory.cs

```
using System;
using System.Collections.Generic;

public class Inventory
{
    private readonly List<InventorySlot> _slots = new List<InventorySlot>();

    public uint SlotCount => (uint) _slots.Count;

    public delegate void SlotUpdateCallback(InventorySlot slot);

    public SlotUpdateCallback OnSlotAdded;
    public SlotUpdateCallback OnSlotRemoved;

    public InventorySlot SelectSlot()
    {
        foreach (var inventory in _slots)
        {
            if (inventory.IsSelected)
            {
                return inventory;
            }
        }
        return null;
    }

    public bool IsSlotSelected()
    {
        foreach (var inventory in _slots)
        {
            if (inventory.IsSelected)
                return inventory.IsSelected;
}

    return false;
  }

  public void ResetAllSelectSlot()
  {
    foreach (var slots in _slots)
    {
      slots.IsSelected = false;
    }
  }

  public InventorySlot CreateSlot()
  {
    InventorySlot newSlot = new InventorySlot();
    _slots.Add(newSlot);

    OnSlotAdded?.Invoke(newSlot);
    return newSlot;
  }

  public void DestroySlot(InventorySlot slot)
  {
    _slots.Remove(slot);
    OnSlotRemoved?.Invoke(slot);
  }

  public void Clear()
  {
    _slots.ForEach(slot => slot.Clear());
  }
```

```
 public void ForEach(Action<InventorySlot> action)
    {
        _slots.ForEach(slot => action(slot));
    }

    public InventorySlot FindFirst(Predicate<InventorySlot> predicate)
    {
        return _slots.Find(predicate);
    }

    public List<InventorySlot> FindAll(Predicate<InventorySlot> predicate)
    {
        return _slots.FindAll(predicate);
    }
}
```

InventorySlot.cs

```
using System;
using System.Collections.Generic;

public class FailedToMoveItemToSlotException : Exception
{
}

public class InventorySlot
{
    public delegate void ItemChangeCallback(InventorySlot slot);

    public ItemChangeCallback OnItemChange;

    private InventoryItem _item;
    private uint _quantity;
    private uint _maxQuantity = uint.MaxValue;
    private List<InventoryItemType> _allowedItemTypes = new List<InventoryItemType>();
```

```
public InventoryItem Item => _item;
public uint Quantity => _quantity;

private bool _isSelected = false;

public bool IsSelected
{
   get => _isSelected;
   set => _isSelected = value;
}
public uint MaxQuantity
{
   get => _maxQuantity;
   set => _maxQuantity = value;
}



public void AddAllowedItemType(InventoryItemType itemType)
{
   _allowedItemTypes.Add(itemType);
}

public void StoreItem(InventoryItem item, uint quantity)
{
   if ((_item == null || _item == item) && CanSlotContainItem(item) && CanAddItemsToSlot(quantity))
   {
      _item = item;
      _quantity += quantity;
      OnItemChange?.Invoke(this);
   }
   else
   {
      throw new FailedToMoveItemToSlotException();
```

```
    }
    }

    public void Clear()
    {
        _item = null;
        _quantity = 0;
        OnItemChange?.Invoke(this);
    }

    public void MoveAllTo(InventorySlot slotDestination)
    {
        MoveTo(slotDestination, _quantity);
    }

    public void MoveTo(InventorySlot slotDestination, uint quantity)
    {
        if (slotDestination == null || quantity > _quantity || !CanSlotContainItem(slotDestination._item) ||
!slotDestination.CanSlotContainItem(_item))
        {
            throw new FailedToMoveItemToSlotException();
        }
        else
        {
            if (slotDestination._item == _item || slotDestination._item == null)
            {
                uint movableQuantity = Math.Min(quantity, slotDestination.MaxQuantity - slotDestination.Quantity);
                slotDestination._item = _item;
                slotDestination._quantity += movableQuantity;
                _quantity -= movableQuantity;

                if (_quantity == 0)
                {
```

```
            Clear();
          }
        }
        else if (_quantity == quantity)
        {
          if (CanSlotHoldItems(slotDestination._quantity) && slotDestination.CanSlotHoldItems(_quantity))
          {
            Utils.Swap(ref slotDestination._item, ref _item);
            Utils.Swap(ref slotDestination._quantity, ref _quantity);
          }
          else
          {
            throw new FailedToMoveItemToSlotException();
          }
        }
        Else
        {
          throw new FailedToMoveItemToSlotException();
        }
      }

      OnItemChange?.Invoke(this);
      slotDestination.OnItemChange?.Invoke(slotDestination);
    }

    private bool CanSlotContainItem(InventoryItem item) //Может Ли Слот Содержать Элемент
    {
      return item == null || CanSlotContainItemType(item.ItemType);
    }

    public bool CanSlotContainItemType(InventoryItemType itemType)
    {
      return _allowedItemTypes.Count == 0 || _allowedItemTypes.Contains(itemType);
```

```
 }

   private bool CanAddItemsToSlot(uint quantity)
   {
      return CanSlotHoldItems(_quantity + quantity);
   }

   private bool CanSlotHoldItems(uint quantity)
   {
      return quantity <= _maxQuantity;
   }
}
```

InventoryItem.cs

```
using UnityEngine;

[CreateAssetMenu(menuName = "ScriptableObject/Inventory/InventoryItem")]
public class InventoryItem : CompositeScriptableObject
{
   [SerializeField] private string _name;
   [SerializeField] private Sprite _sprite;
   [SerializeField] private InventoryItemType _itemType;

   public string Name => _name;
   public Sprite Sprite => _sprite;
   public InventoryItemType ItemType => _itemType;
}
```

Приложение 2

Тест InventorySlotTest

```
using NUnit.Framework;
using UnityEngine;

public class InventorySlotTest
```

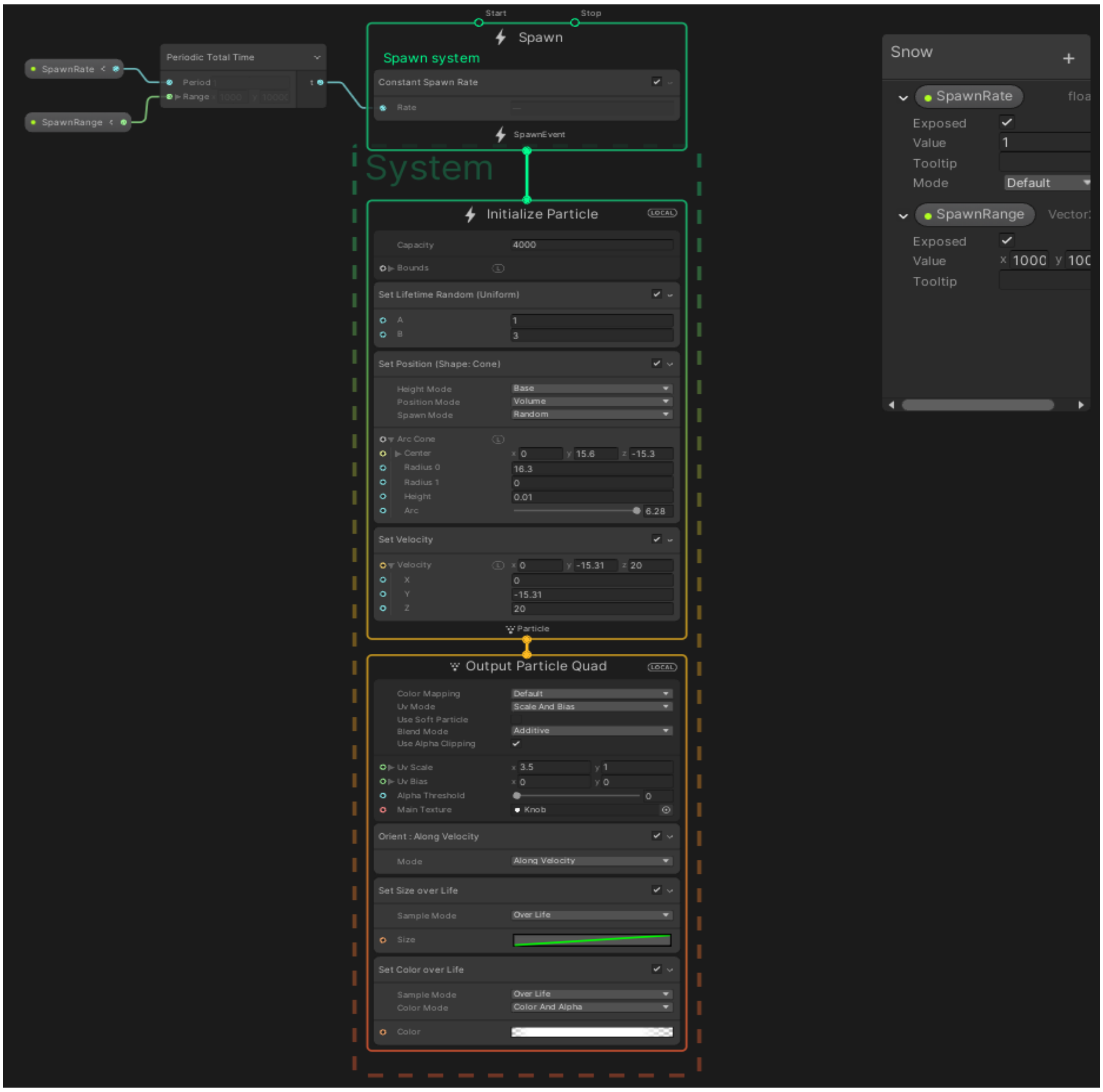
```
{
  [Test]
  public void StoreAndClearSlot()
  {
    InventoryItem testItem = ScriptableObject.CreateInstance<InventoryItem>();

    InventorySlot slot = new InventorySlot();

    slot.StoreItem(testItem, 10);

    Assert.AreEqual(slot.Item, testItem);
    Assert.AreEqual(slot.Quantity, 10);

    slot.Clear();

    Assert.AreEqual(slot.Item, null);
    Assert.AreEqual(slot.Quantity, 0);
  }

  [Test]
  public void MoveToEmptySlot()
  {
    InventoryItem testItem = ScriptableObject.CreateInstance<InventoryItem>();

    InventorySlot slotSource = new InventorySlot();
    InventorySlot slotDestination = new InventorySlot();

    slotSource.StoreItem(testItem, 10);

    slotSource.MoveTo(slotDestination, 4);



    Assert.AreEqual(slotSource.Item, testItem);
    Assert.AreEqual(slotSource.Quantity, 6);
    Assert.AreEqual(slotDestination.Item, testItem);
    Assert.AreEqual(slotDestination.Quantity, 4);
}

[Test]
public void MoveError()
{
    InventoryItem testItem1 = ScriptableObject.CreateInstance<InventoryItem>();
    InventoryItem testItem2 = ScriptableObject.CreateInstance<InventoryItem>();

    InventorySlot slotSource = new InventorySlot();
    InventorySlot slotDestination = new InventorySlot();

    slotSource.StoreItem(testItem1, 10);
    slotDestination.StoreItem(testItem2, 10);

    bool succeeded = false;

    try
    {
        slotSource.MoveTo(slotDestination, 4);
    }
    catch
    {
        succeeded = true;
    }

    Assert.IsTrue(succeeded);
}

[Test]
public void MoveAdd()
```

```
{
    InventoryItem testItem = ScriptableObject.CreateInstance<InventoryItem>();

    InventorySlot slotSource = new InventorySlot();
    InventorySlot slotDestination = new InventorySlot();

    slotSource.StoreItem(testItem, 2);
    slotDestination.StoreItem(testItem, 4);

    slotSource.MoveAllTo(slotDestination);

    Assert.AreEqual(slotSource.Item, null);
    Assert.AreEqual(slotSource.Quantity, 0);
    Assert.AreEqual(slotDestination.Item, testItem);
    Assert.AreEqual(slotDestination.Quantity, 6);
}

[Test]
public void MoveWithExchange()
{
    InventoryItem testItem1 = ScriptableObject.CreateInstance<InventoryItem>();
    InventoryItem testItem2 = ScriptableObject.CreateInstance<InventoryItem>();

    InventorySlot slotSource = new InventorySlot();
    InventorySlot slotDestination = new InventorySlot();
    slotSource.StoreItem(testItem1, 2);
    slotDestination.StoreItem(testItem2, 4);

    slotSource.MoveAllTo(slotDestination);

    Assert.AreEqual(slotSource.Item, testItem2);
    Assert.AreEqual(slotSource.Quantity, 4);
    Assert.AreEqual(slotDestination.Item, testItem1);
```

```
        Assert.AreEqual(slotDestination.Quantity, 2);
    }
}
```

Приложение 3

VFX Снега



Periodic Total Time
Period 1
Range x 1000 y 10000

Spawn
Spawn system
Constant Spawn Rate ✓
Rate
SpawnEvent

Initialize Particle (LOCAL)
Capacity 4000
Bounds
Set Lifetime Random (Uniform) ✓
A 1
B 3
Set Position (Shape: Cone) ✓
Height Mode Base
Position Mode Volume
Spawn Mode Random
Arc Cone
Center x 0 y 15.6 z -15.3
Radius 0 16.3
Radius 1 0
Height 0.01
Arc 6.28
Set Velocity ✓
Velocity x 0 y -15.31 z 20
X 0
Y -15.31
Z 20
Particle

Output Particle Quad (LOCAL)
Color Mapping Default
Uv Mode Scale And Bias
Use Soft Particle
Blend Mode Additive
Use Alpha Clipping ✓
Uv Scale x 3.5 y 1
Uv Bias x 0 y 0
Alpha Threshold 0
Main Texture Knob
Orient : Along Velocity ✓
Mode Along Velocity
Set Size over Life ✓
Sample Mode Over Life
Size
Set Color over Life ✓
Sample Mode Over Life
Color Mode Color And Alpha
Color

Snow
SpawnRate floa
Exposed ✓
Value 1
Tooltip
Mode Default
SpawnRange Vector
Exposed ✓
Value x 1000 y 100
Tooltip