using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Main script to control all AI states

/// </summary>

public class AiBehavior : MonoBehaviour

{

// Navigation agent if it is needed

[HideInInspector]

public NavAgent navAgent;

// This state will be activate on start

public AiState defaultState;

// List with all states for this AI

private List<AiState> aiStates = new List<AiState>();

// The state that was before

private AiState previousState;

// Active state

private AiState currentState;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

if (navAgent == null)

{

// Try to find navigation agent for this object

navAgent = GetComponentInChildren<NavAgent>();

}

}

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

// Enable AI on AiBehavior enabling

if (currentState != null && currentState.enabled == false)

{

EnableNewState();

}

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

// Disable AI on AiBehavior disabling

DisableAllStates();

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

// Get all AI states from this gameobject

AiState[] states = GetComponents<AiState>();

if (states.Length > 0)

{

foreach (AiState state in states)

{

// Add state to list

aiStates.Add(state);

}

if (defaultState != null)

{

// Set active and previous states as default state

previousState = currentState = defaultState;

if (currentState != null)

{

// Go to active state

ChangeState(currentState);

}

else

{

Debug.LogError("Incorrect default AI state " + defaultState);

}

}

else

{

Debug.LogError("AI have no default state");

}

}

else

{

Debug.LogError("No AI states found");

}

}

/// <summary>

/// Set AI to defalt state.

/// </summary>

public void GoToDefaultState()

{

previousState = currentState;

currentState = defaultState;

NotifyOnStateExit();

DisableAllStates();

EnableNewState();

NotifyOnStateEnter();

}

/// <summary>

/// Change Ai state.

/// </summary>

/// <param name="state">State.</param>

public void ChangeState(AiState state)

{

if (state != null)

{

// Try to find such state in list

foreach (AiState aiState in aiStates)

{

if (state == aiState)

{

previousState = currentState;

currentState = aiState;

NotifyOnStateExit();

DisableAllStates();

EnableNewState();

NotifyOnStateEnter();

return;

}

}

Debug.Log("No such state " + state);

// If have no such state - go to default state

GoToDefaultState();

Debug.Log("Go to default state " + aiStates[0]);

}

}

/// <summary>

/// Turn off all AI states components.

/// </summary>

private void DisableAllStates()

{

foreach (AiState aiState in aiStates)

{

aiState.enabled = false;

}

}

/// <summary>

/// Turn on active AI state component.

/// </summary>

private void EnableNewState()

{

currentState.enabled = true;

}

/// <summary>

/// Send OnStateExit notification to previous state.

/// </summary>

private void NotifyOnStateExit()

{

previousState.OnStateExit(previousState, currentState);

}

/// <summary>

/// Send OnStateEnter notification to new state.

/// </summary>

private void NotifyOnStateEnter()

{

currentState.OnStateEnter(previousState, currentState);

}

/// <summary>

/// Raises the trigger event.

/// </summary>

/// <param name="trigger">Trigger.</param>

/// <param name="my">My.</param>

/// <param name="other">Other.</param>

public void OnTrigger(AiState.Trigger trigger, Collider2D my, Collider2D other)

{

if (currentState == null)

{

Debug.Log("Current sate is null");

}

currentState.OnTrigger(trigger, my, other);

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Moving and turning operation.

/// </summary>

public class NavAgent : MonoBehaviour

{

// Speed im m/s

public float speed = 0.75f;

// Can moving

[HideInInspector]

public bool move = true;

// Can turning

[HideInInspector]

public bool turn = true;

// Destination position

[HideInInspector]

public Vector2 destination;

// Velocity vector

[HideInInspector]

public Vector2 velocity;

private Animator anim;

// Position on last frame

private Vector2 prevPosition;

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

prevPosition = transform.position;

anim = GetComponentInParent<Animator>();

}

/// <summary>

/// Update this instance.

/// </summary>

void FixedUpdate()

{

// If moving is allowed

if (move == true)

{

// Move towards destination point

transform.position = Vector2.MoveTowards(transform.position, destination, speed \* Time.fixedDeltaTime);

}

// Calculate velocity

velocity = (Vector2)transform.position - prevPosition;

velocity /= Time.fixedDeltaTime;

// If turning is allowed

if (turn == true)

{

SetSpriteDirection(destination - (Vector2)transform.position);

}

// Save last position

prevPosition = transform.position;

}

/// <summary>

/// Sets sprite direction on x axis.

/// </summary>

/// <param name="direction">Direction.</param>

private void SetSpriteDirection(Vector2 direction)

{

// Flip gameobject dependings on direction

if (direction.x > 0f && transform.localScale.x < 0f) // To the right

{

transform.localScale = new Vector3(-transform.localScale.x, transform.localScale.y, transform.localScale.z);

}

else if (direction.x < 0f && transform.localScale.x > 0f) // To the left

{

transform.localScale = new Vector3(-transform.localScale.x, transform.localScale.y, transform.localScale.z);

}

// Set direction for 8d animation (if used coresponding animator)

if (anim != null)

{

foreach (AnimatorControllerParameter param in anim.parameters)

{

if (param.name == "directionY")

{

float directionY = direction.y / (Mathf.Abs(direction.x) + Mathf.Abs(direction.y));

anim.SetFloat("directionY", directionY);

break;

}

}

}

}

/// <summary>

/// Looks at direction.

/// </summary>

/// <param name="direction">Direction.</param>

public void LookAt(Vector2 direction)

{

SetSpriteDirection(direction);

}

/// <summary>

/// Looks at target.

/// </summary>

/// <param name="target">Target.</param>

public void LookAt(Transform target)

{

SetSpriteDirection(target.position - transform.position);

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Basic class for attacks types.

/// </summary>

public class Attack : MonoBehaviour

{

// Damage amount

public int damage = 1;

// Cooldown between attacks

public float cooldown = 0.5f;

// Delay for fire animation

public float fireDelay = 0f;

// Sound effect

public AudioClip sfx;

public virtual void TryAttack(Transform target)

{

}

public virtual void Fire(Transform target)

{

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Attack with melee weapon

/// </summary>

public class AttackMelee : Attack

{

// Animation controller for this AI

private Animator anim;

// Counter for cooldown calculation

private float cooldownCounter;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

anim = GetComponentInParent<Animator>();

cooldownCounter = cooldown;

}

/// <summary>

/// Update this instance.

/// </summary>

void FixedUpdate()

{

if (cooldownCounter < cooldown)

{

cooldownCounter += Time.fixedDeltaTime;

}

}

/// <summary>

/// Attack the specified target if cooldown expired

/// </summary>

/// <param name="target">Target.</param>

public override void TryAttack(Transform target)

{

if (cooldownCounter >= cooldown)

{

cooldownCounter = 0f;

Fire(target);

}

}

private IEnumerator FireCoroutine(Transform target)

{

if (target != null)

{

// If unit has animator

if (anim != null && anim.runtimeAnimatorController != null)

{

// Search for clip

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Attack")

{

// Play animation

anim.SetTrigger("attack");

break;

}

}

}

// Delay to synchronize with animation

yield return new WaitForSeconds(fireDelay);

if (target != null)

{

// If target can receive damage

DamageTaker damageTaker = target.GetComponent<DamageTaker>();

if (damageTaker != null)

{

damageTaker.TakeDamage(damage);

}

// Play sound effect

if (sfx != null && AudioManager.instance != null)

{

AudioManager.instance.PlayAttack(sfx);

}

}

}

}

/// <summary>

/// Make melee attack

/// </summary>

/// <param name="target">Target.</param>

public override void Fire(Transform target)

{

StartCoroutine(FireCoroutine(target));

}

void OnDestroy()

{

StopAllCoroutines();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Attack with ranged weapon

/// </summary>

public class AttackRanged : Attack

{

// Prefab for arrows

public GameObject arrowPrefab;

// From this position arrows will fired

public Transform firePoint;

// Animation controller for this AI

private Animator anim;

// Counter for cooldown calculation

private float cooldownCounter;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

anim = GetComponentInParent<Animator>();

cooldownCounter = cooldown;

Debug.Assert(arrowPrefab && firePoint, "Wrong initial parameters");

}

/// <summary>

/// Update this instance.

/// </summary>

void FixedUpdate()

{

if (cooldownCounter < cooldown)

{

cooldownCounter += Time.fixedDeltaTime;

}

}

/// <summary>

/// Attack the specified target if cooldown expired

/// </summary>

/// <param name="target">Target.</param>

public override void TryAttack(Transform target)

{

if (cooldownCounter >= cooldown)

{

cooldownCounter = 0f;

Fire(target);

}

}

private IEnumerator FireCoroutine(Transform target, GameObject bulletPrefab)

{

if (target != null && bulletPrefab != null)

{

// If unit has animator

if (anim != null && anim.runtimeAnimatorController != null)

{

// Search for clip

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Attack")

{

// Play animation

anim.SetTrigger("attack");

break;

}

}

}

// Delay to synchronize with animation

yield return new WaitForSeconds(fireDelay);

if (target != null)

{

// Create arrow

GameObject arrow = Instantiate(bulletPrefab, firePoint.position, firePoint.rotation);

IBullet bullet = arrow.GetComponent<IBullet>();

bullet.SetDamage(damage);

bullet.Fire(target);

// Play sound effect

if (sfx != null && AudioManager.instance != null)

{

AudioManager.instance.PlayAttack(sfx);

}

}

}

}

/// <summary>

/// Make ranged attack

/// </summary>

/// <param name="target">Target.</param>

public override void Fire(Transform target)

{

StartCoroutine(FireCoroutine(target, arrowPrefab));

}

/// <summary>

/// Make ranged attack with special bullet

/// </summary>

/// <param name="target">Target.</param>

/// <param name="bulletPrefab">Bullet prefab.</param>

public void Fire(Transform target, GameObject bulletPrefab)

{

StartCoroutine(FireCoroutine(target, bulletPrefab));

}

void OnDestroy()

{

StopAllCoroutines();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System;

/// <summary>

/// Basic class for AI state.

/// </summary>

public class AiState : MonoBehaviour

{

// Allowed triiger types for AI state transactions

public enum Trigger

{

TriggerEnter, // On collider enter

TriggerStay, // On collider stay

TriggerExit, // On collider exit

Damage, // On damage taken

Cooldown, // On some cooldown expired

Alone // When no other colliders intersect during time

}

[Serializable]

// Allows to specify AI state change on any trigger

public class AiTransaction

{

public Trigger trigger;

public AiState newState;

}

// List with specified transactions for this AI state

public AiTransaction[] specificTransactions;

// Animation controller for this AI

protected Animator anim;

// AI behavior of this object

protected AiBehavior aiBehavior;

/// <summary>

/// Awake this instance.

/// </summary>

public virtual void Awake()

{

aiBehavior = GetComponent<AiBehavior> ();

anim = GetComponentInParent<Animator>();

Debug.Assert (aiBehavior, "Wrong initial parameters");

}

/// <summary>

/// Raises the state enter event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public virtual void OnStateEnter(AiState previousState, AiState newState)

{

}

/// <summary>

/// Raises the state exit event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public virtual void OnStateExit(AiState previousState, AiState newState)

{

}

/// <summary>

/// Raises the trigger event.

/// </summary>

/// <param name="trigger">Trigger.</param>

/// <param name="my">My.</param>

/// <param name="other">Other.</param>

public virtual bool OnTrigger(Trigger trigger, Collider2D my, Collider2D other)

{

bool res = false;

// Check if this AI state has specific transactions for this trigger

foreach (AiTransaction transaction in specificTransactions)

{

if (trigger == transaction.trigger)

{

aiBehavior.ChangeState(transaction.newState);

res = true;

break;

}

}

return res;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Allows AI to attack targets.

/// </summary>

public class AiStateAttack : AiState

{

[Space(10)]

// Attack target closest to the capture point

public bool useTargetPriority = false;

// Go to this state if passive event occures

public AiState passiveAiState;

// Target for attack

private GameObject target;

// List with potential targets finded during this frame

private List<GameObject> targetsList = new List<GameObject>();

// My melee attack type if it is

private Attack meleeAttack;

// My ranged attack type if it is

private Attack rangedAttack;

// Type of last attack is made

private Attack myLastAttack;

// Allows to await new target for one frame before exit from this state

private bool targetless;

/// <summary>

/// Awake this instance.

/// </summary>

public override void Awake()

{

base.Awake();

meleeAttack = GetComponentInChildren<AttackMelee>() as Attack;

rangedAttack = GetComponentInChildren<AttackRanged>() as Attack;

}

/// <summary>

/// Raises the state enter event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateEnter(AiState previousState, AiState newState)

{

// Stop to moving

if (aiBehavior.navAgent != null)

{

aiBehavior.navAgent.move = false;

}

}

/// <summary>

/// Raises the state exit event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateExit(AiState previousState, AiState newState)

{

LoseTarget();

}

/// <summary>

/// FixedUpdate for this instance.

/// </summary>

void FixedUpdate()

{

// If have no target - try to get new target

if ((target == null) && (targetsList.Count > 0))

{

target = GetTopmostTarget();

}

// There are no targets around

if (target == null)

{

if (targetless == false)

{

targetless = true;

}

else

{

// If have no target more than one frame - exit from this state

aiBehavior.ChangeState(passiveAiState);

}

}

}

/// <summary>

/// Gets top priority target from list.

/// </summary>

/// <returns>The topmost target.</returns>

private GameObject GetTopmostTarget()

{

GameObject res = null;

if (useTargetPriority == true) // Get target with minimum distance to capture point

{

float minPathDistance = float.MaxValue;

foreach (GameObject ai in targetsList)

{

if (ai != null)

{

AiStatePatrol aiStatePatrol = ai.GetComponent<AiStatePatrol>();

float distance = aiStatePatrol.GetRemainingPath();

if (distance < minPathDistance)

{

minPathDistance = distance;

res = ai;

}

}

}

}

else // Get first target from list

{

res = targetsList[0];

}

// Clear list of potential targets

targetsList.Clear();

return res;

}

/// <summary>

/// Loses the current target.

/// </summary>

private void LoseTarget()

{

target = null;

targetless = false;

myLastAttack = null;

}

/// <summary>

/// Raises the trigger event.

/// </summary>

/// <param name="trigger">Trigger.</param>

/// <param name="my">My.</param>

/// <param name="other">Other.</param>

public override bool OnTrigger(AiState.Trigger trigger, Collider2D my, Collider2D other)

{

if (base.OnTrigger(trigger, my, other) == false)

{

switch (trigger)

{

case AiState.Trigger.TriggerStay:

TriggerStay(my, other);

break;

case AiState.Trigger.TriggerExit:

TriggerExit(my, other);

break;

}

}

return false;

}

/// <summary>

/// Triggers the stay.

/// </summary>

/// <param name="my">My.</param>

/// <param name="other">Other.</param>

private void TriggerStay(Collider2D my, Collider2D other)

{

if (target == null) // Add new target to potential targets list

{

targetsList.Add(other.gameObject);

}

else // Attack current target

{

// If this is my current target

if (target == other.gameObject)

{

if (my.name == "MeleeAttack") // If target is in melee attack range

{

// If I have melee attack type

if (meleeAttack != null)

{

if (aiBehavior.navAgent != null)

{

// Look at target

aiBehavior.navAgent.LookAt(target.transform);

}

// Remember my last attack type

myLastAttack = meleeAttack as Attack;

// Try to make melee attack

meleeAttack.TryAttack(other.transform);

}

}

else if (my.name == "RangedAttack") // If target is in ranged attack range

{

// If I have ranged attack type

if (rangedAttack != null)

{

// If target not in melee attack range

if ((meleeAttack == null)

|| ((meleeAttack != null) && (myLastAttack != meleeAttack)))

{

if (aiBehavior.navAgent != null)

{

// Look at target

aiBehavior.navAgent.LookAt(target.transform);

}

// Remember my last attack type

myLastAttack = rangedAttack as Attack;

// Try to make ranged attack

rangedAttack.TryAttack(other.transform);

}

}

}

}

}

}

/// <summary>

/// Triggers the exit.

/// </summary>

/// <param name="my">My.</param>

/// <param name="other">Other.</param>

private void TriggerExit(Collider2D my, Collider2D other)

{

if (other.gameObject == target)

{

// Lose my target if it quit attack range

LoseTarget();

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Allows AI to operate idle state.

/// </summary>

public class AiStateIdle : AiState

{

/// <summary>

/// Raises the state enter event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateEnter(AiState previousState, AiState newState)

{

// Stop to moving and turning

if (aiBehavior.navAgent != null)

{

aiBehavior.navAgent.move = false;

aiBehavior.navAgent.turn = false;

}

// If unit has animator

if (anim != null && anim.runtimeAnimatorController != null)

{

// Search for clip

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Idle")

{

// Play animation

anim.SetTrigger("idle");

break;

}

}

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Allows AI to operate move towards destination.

/// </summary>

public class AiStateMove : AiState

{

[Space(10)]

// Go to this state if passive event occures

public AiState passiveAiState;

// End point for moving

[HideInInspector]

public Transform destination;

/// <summary>

/// Awake this instance.

/// </summary>

public override void Awake()

{

base.Awake();

Debug.Assert (aiBehavior.navAgent, "Wrong initial parameters");

}

/// <summary>

/// Raises the state enter event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateEnter(AiState previousState, AiState newState)

{

// Set destination for navigation agent

aiBehavior.navAgent.destination = destination.position;

// Start moving

aiBehavior.navAgent.move = true;

aiBehavior.navAgent.turn = true;

// If unit has animator

if (anim != null && anim.runtimeAnimatorController != null)

{

// Search for clip

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Move")

{

// Play animation

anim.SetTrigger("move");

break;

}

}

}

}

/// <summary>

/// Raises the state exit event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateExit(AiState previousState, AiState newState)

{

// Stop moving

aiBehavior.navAgent.move = false;

aiBehavior.navAgent.turn = false;

}

/// <summary>

/// Fixed update for this instance.

/// </summary>

void FixedUpdate()

{

// If destination reached

if ((Vector2)transform.position == (Vector2)destination.position)

{

// Look at required direction

aiBehavior.navAgent.LookAt(destination.right);

// Go to passive state

aiBehavior.ChangeState(passiveAiState);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Allows AI to move with specified path.

/// </summary>

public class AiStatePatrol : AiState

{

[Space(10)]

[HideInInspector]

// Specified path

public Pathway path;

// Need to loop path after last point is reached?

public bool loop = false;

[HideInInspector]

// Current destination

public Waypoint destination;

/// <summary>

/// Awake this instance.

/// </summary>

public override void Awake()

{

base.Awake();

Debug.Assert (aiBehavior.navAgent, "Wrong initial parameters");

}

/// <summary>

/// Raises the state enter event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateEnter(AiState previousState, AiState newState)

{

if (path == null)

{

// If I have no path - try to find it

path = FindObjectOfType<Pathway>();

Debug.Assert(path, "Have no path");

}

if (destination == null)

{

// Get next waypoint from my path

destination = path.GetNearestWaypoint(transform.position);

}

// Set destination for navigation agent

aiBehavior.navAgent.destination = destination.transform.position;

// Start moving

aiBehavior.navAgent.move = true;

aiBehavior.navAgent.turn = true;

// If unit has animator

if (anim != null && anim.runtimeAnimatorController != null)

{

// Search for clip

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Move")

{

// Play animation

anim.SetTrigger("move");

break;

}

}

}

}

/// <summary>

/// Raises the state exit event.

/// </summary>

/// <param name="previousState">Previous state.</param>

/// <param name="newState">New state.</param>

public override void OnStateExit(AiState previousState, AiState newState)

{

// Stop moving

aiBehavior.navAgent.move = false;

aiBehavior.navAgent.turn = false;

}

/// <summary>

/// Fixed update for this instance.

/// </summary>

void FixedUpdate()

{

if (destination != null)

{

// If destination reached

if ((Vector2)destination.transform.position == (Vector2)transform.position)

{

// Get next waypoint from my path

destination = path.GetNextWaypoint (destination, loop);

if (destination != null)

{

// Set destination for navigation agent

aiBehavior.navAgent.destination = destination.transform.position;

}

}

}

}

/// <summary>

/// Gets the remaining distance on pathway.

/// </summary>

/// <returns>The remaining path.</returns>

public float GetRemainingPath()

{

Vector2 distance = destination.transform.position - transform.position;

return (distance.magnitude + path.GetPathDistance(destination));

}

/// <summary>

/// Updates the destination.

/// </summary>

/// <param name="getNearestWaypoint">If set to <c>true</c> get nearest waypoint automaticaly.</param>

public void UpdateDestination(bool getNearestWaypoint)

{

if (getNearestWaypoint == true)

{

// Get next waypoint from my path

destination = path.GetNearestWaypoint(transform.position);

}

if (enabled == true)

{

// Set destination for navigation agent

aiBehavior.navAgent.destination = destination.transform.position;

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class AiTriggerAlone : MonoBehaviour

{

// Alone time before trigger

public float aloneDuration = 1f;

public List<Component> receivers = new List<Component>();

// Allowed objects tags for collision detection

public List<string> tags = new List<string>();

// My AiBehavior

private AiBehavior ai;

// My collider

private Collider2D col;

// Alone duration countrer

private float counter;

// Already triggered

private bool triggered;

void Awake()

{

ai = GetComponentInParent<AiBehavior>();

col = GetComponent<Collider2D>();

Debug.Assert(ai && col, "Wrong initial parameters");

col.enabled = false;

}

void Start()

{

col.enabled = true;

counter = aloneDuration;

triggered = false;

}

/// <summary>

/// Raises the trigger stay2d event.

/// </summary>

/// <param name="other">Other.</param>

void OnTriggerStay2D(Collider2D other)

{

if (other.gameObject != ai.gameObject && IsTagAllowed(other.tag) == true)

{

if (triggered == true)

{

foreach (Component receiver in receivers)

{

receiver.SendMessage("OnTriggerAloneEnd");

}

}

triggered = false;

counter = aloneDuration;

}

}

void FixedUpdate()

{

if (triggered == false)

{

if (counter > 0f)

{

counter -= Time.fixedDeltaTime;

}

else

{

triggered = true;

counter = 0f;

foreach (Component receiver in receivers)

{

receiver.SendMessage("OnTriggerAloneStart");

}

}

}

}

/// <summary>

/// Determines whether this instance is tag allowed the specified tag.

/// </summary>

/// <returns><c>true</c> if this instance is tag allowed the specified tag; otherwise, <c>false</c>.</returns>

/// <param name="tag">Tag.</param>

private bool IsTagAllowed(string tag)

{

bool res = false;

if (tags.Count > 0)

{

foreach (string str in tags)

{

if (str == tag)

{

res = true;

break;

}

}

}

else

{

res = true;

}

return res;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Dynamic filter for AI collision mask

/// </summary>

public class AiTriggerCollider : MonoBehaviour

{

// Allowed objects tags for collision detection

public List<string> tags = new List<string>();

// My collider

private Collider2D col;

// AI behaviour component in parent object

private AiBehavior aiBehavior;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

col = GetComponent<Collider2D>();

aiBehavior = GetComponentInParent<AiBehavior>();

Debug.Assert(col && aiBehavior, "Wrong initial parameters");

col.enabled = false;

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

col.enabled = true;

}

/// <summary>

/// Determines whether this instance is tag allowed the specified tag.

/// </summary>

/// <returns><c>true</c> if this instance is tag allowed the specified tag; otherwise, <c>false</c>.</returns>

/// <param name="tag">Tag.</param>

private bool IsTagAllowed(string tag)

{

bool res = false;

if (tags.Count > 0)

{

foreach (string str in tags)

{

if (str == tag)

{

res = true;

break;

}

}

}

else

{

res = true;

}

return res;

}

/// <summary>

/// Raises the trigger enter2d event.

/// </summary>

/// <param name="other">Other.</param>

void OnTriggerEnter2D(Collider2D other)

{

if (IsTagAllowed(other.tag) == true)

{

// Notify AI behavior about this event

aiBehavior.OnTrigger(AiState.Trigger.TriggerEnter, col, other);

}

}

/// <summary>

/// Raises the trigger stay2d event.

/// </summary>

/// <param name="other">Other.</param>

void OnTriggerStay2D(Collider2D other)

{

if (IsTagAllowed(other.tag) == true)

{

// Notify AI behavior about this event

aiBehavior.OnTrigger(AiState.Trigger.TriggerStay, col, other);

}

}

/// <summary>

/// Raises the trigger exit2d event.

/// </summary>

/// <param name="other">Other.</param>

void OnTriggerExit2D(Collider2D other)

{

if (IsTagAllowed(other.tag) == true)

{

// Notify AI behavior about this event

aiBehavior.OnTrigger(AiState.Trigger.TriggerExit, col, other);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Plays soundtrack and sound effects. Limits number of simultaneously played sound effects.

/// </summary>

public class AudioManager : MonoBehaviour

{

// Singleton

public static AudioManager instance;

// Sound source for sound effects

public AudioSource soundSource;

// Sound source for soundtrack

public AudioSource musicSource;

// Soundtrack

public AudioClip track;

// Wave start sfx

public AudioClip waveStart;

// Enemy reached capture point sfx

public AudioClip captured;

// Player click on tower sfx

public AudioClip towerClick;

// Player click on unit sfx

public AudioClip unitClick;

// Player click UI sfx

public AudioClip uiClick;

// Tower build sfx

public AudioClip towerBuild;

// Tower sell sfx

public AudioClip towerSell;

// Defeat sfx

public AudioClip defeat;

// Victory sfx

public AudioClip victory;

// Attack sfx is played now

private bool attackCoroutine = false;

// Die sfx is played now

private bool dieCoroutine = false;

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

instance = this;

EventManager.StartListening("GamePaused", GamePaused);

EventManager.StartListening("WaveStart", WaveStart);

EventManager.StartListening("Captured", Captured);

EventManager.StartListening("UserClick", UserClick);

EventManager.StartListening("UserUiClick", UserUiClick);

EventManager.StartListening("TowerBuild", TowerBuild);

EventManager.StartListening("TowerSell", TowerSell);

EventManager.StartListening("Defeat", Defeat);

EventManager.StartListening("Victory", Victory);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("GamePaused", GamePaused);

EventManager.StopListening("WaveStart", WaveStart);

EventManager.StopListening("Captured", Captured);

EventManager.StopListening("UserClick", UserClick);

EventManager.StopListening("UserUiClick", UserUiClick);

EventManager.StopListening("TowerBuild", TowerBuild);

EventManager.StopListening("TowerSell", TowerSell);

EventManager.StopListening("Defeat", Defeat);

EventManager.StopListening("Victory", Victory);

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

Debug.Assert(soundSource && musicSource, "Wrong initial settings");

// Set volume from stored configurations

SetVolume(DataManager.instance.configs.soundVolume, DataManager.instance.configs.musicVolume);

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

StopAllCoroutines();

if (instance == this)

{

instance = null;

}

}

/// <summary>

/// Games the paused.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void GamePaused(GameObject obj, string param)

{

if (param == bool.TrueString) // Paused

{

// Pause soundtrack

musicSource.Pause();

}

else // Unpaused

{

// Play soundtrack

if (track != null)

{

musicSource.clip = track;

musicSource.Play();

}

}

}

/// <summary>

/// Sets the volume.

/// </summary>

/// <param name="sound">Sound.</param>

/// <param name="music">Music.</param>

public void SetVolume(float sound, float music)

{

soundSource.volume = sound;

musicSource.volume = music;

}

/// <summary>

/// Plaies the sfx (without limit).

/// </summary>

/// <param name="audioClip">Audio clip.</param>

public void PlaySound(AudioClip audioClip)

{

soundSource.PlayOneShot(audioClip, soundSource.volume);

}

/// <summary>

/// Plaies the attack sfx (one sfx at the same time).

/// </summary>

/// <param name="audioClip">Audio clip.</param>

public void PlayAttack(AudioClip audioClip)

{

if (attackCoroutine == false)

{

StartCoroutine(AttackCoroutine(audioClip));

}

}

/// <summary>

/// Attacks coroutine.

/// </summary>

/// <returns>The coroutine.</returns>

/// <param name="audioClip">Audio clip.</param>

private IEnumerator AttackCoroutine(AudioClip audioClip)

{

attackCoroutine = true;

PlaySound(audioClip);

// Wait for clip end

yield return new WaitForSeconds(audioClip.length);

attackCoroutine = false;

}

/// <summary>

/// Plaies the die sfx (one sfx at the same time).

/// </summary>

/// <param name="audioClip">Audio clip.</param>

public void PlayDie(AudioClip audioClip)

{

if (dieCoroutine == false)

{

StartCoroutine(DieCoroutine(audioClip));

}

}

/// <summary>

/// Dies coroutine.

/// </summary>

/// <returns>The coroutine.</returns>

/// <param name="audioClip">Audio clip.</param>

private IEnumerator DieCoroutine(AudioClip audioClip)

{

dieCoroutine = true;

PlaySound(audioClip);

// Wait for clip end

yield return new WaitForSeconds(audioClip.length);

dieCoroutine = false;

}

/// <summary>

/// Waves started.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void WaveStart(GameObject obj, string param)

{

if (waveStart != null)

{

PlaySound(waveStart);

}

}

/// <summary>

/// Enemy reached capture point.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void Captured(GameObject obj, string param)

{

if (captured != null)

{

PlaySound(captured);

}

}

/// <summary>

/// On user UI click.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UserUiClick(GameObject obj, string param)

{

if (obj != null)

{

PlaySound(uiClick);

}

}

/// <summary>

/// User click handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UserClick(GameObject obj, string param)

{

if (obj != null)

{

Tower tower = obj.GetComponent<Tower>();

if (tower != null)

{

PlaySound(towerClick);

}

else

{

UnitInfo unitInfo = obj.GetComponent<UnitInfo>();

if (unitInfo != null)

{

PlaySound(unitClick);

}

}

}

}

/// <summary>

/// Towers build handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void TowerBuild(GameObject obj, string param)

{

if (towerBuild != null)

{

PlaySound(towerBuild);

}

}

/// <summary>

/// Towers sell handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void TowerSell(GameObject obj, string param)

{

if (towerSell != null)

{

PlaySound(towerSell);

}

}

/// <summary>

/// Defeat handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void Defeat(GameObject obj, string param)

{

if (defeat != null)

{

PlaySound(defeat);

}

}

/// <summary>

/// Victory handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void Victory(GameObject obj, string param)

{

if (victory != null)

{

PlaySound(victory);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Area Of Effect damage on destroing.

/// </summary>

public class AoeDamage : MonoBehaviour

{

// Percent of AOE damage in part of IBullet damage. 0f = 0%, 1f = 100%

public float aoeDamageRate = 1f;

// Area radius

public float radius = 0.3f;

// Explosion prefab

public GameObject explosion;

// Explosion visual duration

public float explosionDuration = 1f;

// Sound effect

public AudioClip sfx;

// Allowed objects tags for collision detection

public List<string> tags = new List<string>();

// IBullet component of this gameObject to get the damage amount

private IBullet bullet;

// Scene is closed now. Forbidden to create new objects on destroy

private bool isQuitting;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

bullet = GetComponent<IBullet>();

Debug.Assert(bullet != null, "Wrong initial settings");

}

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("SceneQuit", SceneQuit);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("SceneQuit", SceneQuit);

}

/// <summary>

/// Raises the application quit event.

/// </summary>

void OnApplicationQuit()

{

isQuitting = true;

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

// If scene is in progress

if (isQuitting == false)

{

// Find all colliders in specified radius

Collider2D[] cols = Physics2D.OverlapCircleAll(transform.position, radius);

foreach (Collider2D col in cols)

{

if (IsTagAllowed(col.tag) == true)

{

// If target can receive damage

DamageTaker damageTaker = col.gameObject.GetComponent<DamageTaker>();

if (damageTaker != null)

{

// Target takes damage equal bullet damage \* AOE Damage Rate

damageTaker.TakeDamage((int)(Mathf.Ceil(aoeDamageRate \* (float)bullet.GetDamage())));

}

}

}

if (explosion != null)

{

// Create explosion visual effect

Destroy(Instantiate<GameObject>(explosion, transform.position, transform.rotation), explosionDuration);

}

if (sfx != null && AudioManager.instance != null)

{

// Play sfx

AudioManager.instance.PlaySound(sfx);

}

}

}

/// <summary>

/// Raises on scene quit.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void SceneQuit(GameObject obj, string param)

{

isQuitting = true;

}

/// <summary>

/// Determines whether this instance is tag allowed the specified tag.

/// </summary>

/// <returns><c>true</c> if this instance is tag allowed the specified tag; otherwise, <c>false</c>.</returns>

/// <param name="tag">Tag.</param>

private bool IsTagAllowed(string tag)

{

bool res = false;

if (tags.Count > 0)

{

foreach (string str in tags)

{

if (str == tag)

{

res = true;

break;

}

}

}

else

{

res = true;

}

return res;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class AoeEffect : MonoBehaviour

{

// Name of effect

public string effectName;

// Effect modifier (-1f = -100%, 0f = 0%, 1f = 100%)

public float modifier = 1f;

// Effect duration

public float duration = 3f;

// Area radius

public float radius = 1f;

// Explosion FX prefab

public GameObject explosionFx;

// Explosion FX duration

public float explosionFxDuration = 1f;

// Effect FX prefab

public GameObject effectFx;

// Sound effect

public AudioClip sfx;

// Allowed objects tags for collision detection

public List<string> tags = new List<string>();

// Scene is closed now. Forbidden to create new objects on destroy

private bool isQuitting;

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("SceneQuit", SceneQuit);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("SceneQuit", SceneQuit);

}

/// <summary>

/// Raises the application quit event.

/// </summary>

void OnApplicationQuit()

{

isQuitting = true;

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

// If scene is in progress

if (isQuitting == false)

{

// Find all colliders in specified radius

Collider2D[] cols = Physics2D.OverlapCircleAll(transform.position, radius);

foreach (Collider2D col in cols)

{

if (IsTagAllowed(col.tag) == true)

{

EffectControl effectControl = col.gameObject.GetComponent<EffectControl>();

if (effectControl != null)

{

effectControl.AddEffect(effectName, modifier, duration, effectFx);

}

}

}

if (explosionFx != null)

{

// Create explosion visual effect

Destroy(Instantiate<GameObject>(explosionFx, transform.position, transform.rotation), explosionFxDuration);

}

if (sfx != null && AudioManager.instance != null)

{

// Play sfx

AudioManager.instance.PlaySound(sfx);

}

}

}

/// <summary>

/// Raises on scene quit.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void SceneQuit(GameObject obj, string param)

{

isQuitting = true;

}

/// <summary>

/// Determines whether this instance is tag allowed the specified tag.

/// </summary>

/// <returns><c>true</c> if this instance is tag allowed the specified tag; otherwise, <c>false</c>.</returns>

/// <param name="tag">Tag.</param>

private bool IsTagAllowed(string tag)

{

bool res = false;

if (tags.Count > 0)

{

foreach (string str in tags)

{

if (str == tag)

{

res = true;

break;

}

}

}

else

{

res = true;

}

return res;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.AI;

/// <summary>

/// Arrow fly trajectory.

/// </summary>

public class BulletArrow : MonoBehaviour, IBullet

{

// Damage amount

[HideInInspector] int damage = 1;

// Maximum life time

public float lifeTime = 3f;

// Starting speed

public float speed = 3f;

// Constant acceleration

public float speedUpOverTime = 0.5f;

// If target is close than this distance - it will be hitted

public float hitDistance = 0.2f;

// Ballistic trajectory offset (in distance to target)

public float ballisticOffset = 0.5f;

// Do not rotate bullet during fly

public bool freezeRotation = false;

// This bullet don't deal damage to single target. Only AOE damage if it is

public bool aoeDamageOnly = false;

// From this position bullet was fired

private Vector2 originPoint;

// Aimed target

private Transform target;

// Last target's position

private Vector2 aimPoint;

// Current position without ballistic offset

private Vector2 myVirtualPosition;

// Position on last frame

private Vector2 myPreviousPosition;

// Counter for acceleration calculation

private float counter;

// Image of this bullet

private SpriteRenderer sprite;

/// <summary>

/// Set damage amount for this bullet.

/// </summary>

/// <param name="damage">Damage.</param>

public void SetDamage(int damage)

{

this.damage = damage;

}

/// <summary>

/// Get damage amount for this bullet.

/// </summary>

/// <returns>The damage.</returns>

public int GetDamage()

{

return damage;

}

/// <summary>

/// Fire bullet towards specified target.

/// </summary>

/// <param name="target">Target.</param>

public void Fire(Transform target)

{

sprite = GetComponent<SpriteRenderer>();

// Disable sprite on first frame beqause we do not know fly direction yet

sprite.enabled = false;

originPoint = myVirtualPosition = myPreviousPosition = transform.position;

this.target = target;

aimPoint = target.position;

// Destroy bullet after lifetime

Destroy(gameObject, lifeTime);

}

/// <summary>

/// Update this instance.

/// </summary>

void FixedUpdate ()

{

counter += Time.fixedDeltaTime;

// Add acceleration

speed += Time.fixedDeltaTime \* speedUpOverTime;

if (target != null)

{

aimPoint = target.position;

}

// Calculate distance from firepoint to aim

Vector2 originDistance = aimPoint - originPoint;

// Calculate remaining distance

Vector2 distanceToAim = aimPoint - (Vector2)myVirtualPosition;

// Move towards aim

myVirtualPosition = Vector2.Lerp(originPoint, aimPoint, counter \* speed / originDistance.magnitude);

// Add ballistic offset to trajectory

transform.position = AddBallisticOffset(originDistance.magnitude, distanceToAim.magnitude);

// Rotate bullet towards trajectory

LookAtDirection2D((Vector2)transform.position - myPreviousPosition);

myPreviousPosition = transform.position;

sprite.enabled = true;

// Close enough to hit

if (distanceToAim.magnitude <= hitDistance)

{

if (target != null)

{

// If bullet must deal damage to single target

if (aoeDamageOnly == false)

{

// If target can receive damage

DamageTaker damageTaker = target.GetComponent<DamageTaker>();

if (damageTaker != null)

{

damageTaker.TakeDamage(damage);

}

}

}

// Destroy bullet

Destroy(gameObject);

}

}

/// <summary>

/// Looks at direction2d.

/// </summary>

/// <param name="direction">Direction.</param>

private void LookAtDirection2D(Vector2 direction)

{

if (freezeRotation == false)

{

float angle = Mathf.Atan2(direction.y, direction.x) \* Mathf.Rad2Deg;

transform.rotation = Quaternion.AngleAxis(angle, Vector3.forward);

}

}

/// <summary>

/// Adds ballistic offset to trajectory.

/// </summary>

/// <returns>The ballistic offset.</returns>

/// <param name="originDistance">Origin distance.</param>

/// <param name="distanceToAim">Distance to aim.</param>

private Vector2 AddBallisticOffset(float originDistance, float distanceToAim)

{

if (ballisticOffset > 0f)

{

// Calculate sinus offset

float offset = Mathf.Sin(Mathf.PI \* ((originDistance - distanceToAim) / originDistance));

offset \*= originDistance;

// Add offset to trajectory

return (Vector2)myVirtualPosition + (ballisticOffset \* offset \* Vector2.up);

}

else

{

return myVirtualPosition;

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Camera moving and autoscaling.

/// </summary>

[RequireComponent(typeof(Camera))]

public class CameraControl : MonoBehaviour

{

/// <summary>

/// Control type for camera autoscaling.

/// </summary>

public enum ControlType

{

ConstantWidth, // Camera will keep constant width

ConstantHeight, // Camera will keep constant height

OriginCameraSize // Do not scale camera

}

// Camera control type

public ControlType controlType;

// Camera will autoscale to fit this object

public SpriteRenderer focusObjectRenderer;

// Horizontal offset from focus object edges

public float offsetX = 0f;

// Vertical offset from focus object edges

public float offsetY = 0f;

// Camera speed when moving (dragging)

public float dragSpeed = 2f;

// Restrictive points for camera moving

private float maxX, minX, maxY, minY;

// Camera dragging at now vector

private float moveX, moveY;

// Camera component from this gameobject

private Camera cam;

// Origin camera aspect ratio

private float originAspect;

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

cam = GetComponent<Camera>();

Debug.Assert(focusObjectRenderer && cam, "Wrong initial settings");

originAspect = cam.aspect;

// Get restrictive points from focus object's corners

maxX = focusObjectRenderer.bounds.max.x;

minX = focusObjectRenderer.bounds.min.x;

maxY = focusObjectRenderer.bounds.max.y;

minY = focusObjectRenderer.bounds.min.y;

UpdateCameraSize();

}

/// <summary>

/// Lates update this instance.

/// </summary>

void LateUpdate()

{

// Camera aspect ratio is changed

if (originAspect != cam.aspect)

{

UpdateCameraSize();

originAspect = cam.aspect;

}

// Need to move camera horizontally

if (moveX != 0f)

{

bool permit = false;

// Move to right

if (moveX > 0f)

{

// If restrictive point does not reached

if (cam.transform.position.x + (cam.orthographicSize \* cam.aspect) < maxX - offsetX)

{

permit = true;

}

}

// Move to left

else

{

// If restrictive point does not reached

if (cam.transform.position.x - (cam.orthographicSize \* cam.aspect) > minX + offsetX)

{

permit = true;

}

}

if (permit == true)

{

// Move camera

transform.Translate(Vector3.right \* moveX \* dragSpeed, Space.World);

}

moveX = 0f;

}

// Need to move camera vertically

if (moveY != 0f)

{

bool permit = false;

// Move up

if (moveY > 0f)

{

// If restrictive point does not reached

if (cam.transform.position.y + cam.orthographicSize < maxY - offsetY)

{

permit = true;

}

}

// Move down

else

{

// If restrictive point does not reached

if (cam.transform.position.y - cam.orthographicSize > minY + offsetY)

{

permit = true;

}

}

if (permit == true)

{

// Move camera

transform.Translate (Vector3.up \* moveY \* dragSpeed, Space.World);

}

moveY = 0f;

}

}

/// <summary>

/// Need to move camera horizontally.

/// </summary>

/// <param name="distance">Distance.</param>

public void MoveX(float distance)

{

moveX = distance;

}

/// <summary>

/// Need to move camera vertically.

/// </summary>

/// <param name="distance">Distance.</param>

public void MoveY(float distance)

{

moveY = distance;

}

/// <summary>

/// Updates the size of the camera to fit focus object.

/// </summary>

private void UpdateCameraSize()

{

switch (controlType)

{

case ControlType.ConstantWidth:

cam.orthographicSize = (maxX - minX - 2 \* offsetX) / (2f \* cam.aspect);

break;

case ControlType.ConstantHeight:

cam.orthographicSize = (maxY - minY - 2 \* offsetY) / 2f;

break;

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

/// <summary>

/// Version of saved data format. Use it to check if stored data format is equal to actual data format

/// </summary>

[Serializable]

public class DataVersion

{

public int major = 1;

public int minor = 0;

}

/// <summary>

/// Format of stored game progress data.

/// </summary>

[Serializable]

public class GameProgressData

{

public System.DateTime saveTime = DateTime.MinValue; // Saving time

public string lastCompetedLevel = ""; // Name of level was last completed

public List<string> openedLevels = new List<string>(); // List with levels available to play

}

/// <summary>

/// Format of stored game configurations.

/// </summary>

[Serializable]

public class GameConfigurations

{

public float soundVolume = 0.5f;

public float musicVolume = 0.5f;

}

/// <summary>

/// Saving and load data from file.

/// </summary>

public class DataManager : MonoBehaviour

{

// Singleton

public static DataManager instance;

// Game progress data container

public GameProgressData progress = new GameProgressData();

// Game configurations container

public GameConfigurations configs = new GameConfigurations();

// Data version container

private DataVersion dataVersion = new DataVersion();

// Name of file with data version

private string dataVersionFile = "/DataVersion.dat";

// Name of file with game progress data

private string gameProgressFile = "/GameProgress.dat";

// Name of file with game configurations

private string gameConfigsFile = "/GameConfigs.dat";

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

if (instance == null)

{

instance = this;

DontDestroyOnLoad(gameObject);

UpdateDataVersion();

LoadGameProgress();

LoadGameConfigs();

}

else if (instance != this)

{

Destroy(gameObject);

return;

}

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

if (instance == this)

{

instance = null;

}

}

/// <summary>

/// Updates the version of data format.

/// </summary>

private void UpdateDataVersion()

{

if (File.Exists(Application.persistentDataPath + dataVersionFile) == true)

{

BinaryFormatter bfOpen = new BinaryFormatter();

FileStream fileToOpen = File.Open(Application.persistentDataPath + dataVersionFile, FileMode.Open);

DataVersion version = (DataVersion)bfOpen.Deserialize(fileToOpen);

fileToOpen.Close();

switch (version.major)

{

case 1:

// Stored data has version 1.x

// Some handler to convert data if it is needed ...

break;

}

}

BinaryFormatter bfCreate = new BinaryFormatter();

FileStream fileToCreate = File.Create(Application.persistentDataPath + dataVersionFile);

bfCreate.Serialize(fileToCreate, dataVersion);

fileToCreate.Close();

}

/// <summary>

/// Delete file with saved game data. For debug only

/// </summary>

public void DeleteGameProgress()

{

File.Delete(Application.persistentDataPath + gameProgressFile);

progress = new GameProgressData();

Debug.Log("Saved game progress deleted");

}

/// <summary>

/// Saves the game progress to file.

/// </summary>

public void SaveGameProgress()

{

BinaryFormatter bf = new BinaryFormatter();

FileStream file = File.Create(Application.persistentDataPath + gameProgressFile);

progress.saveTime = DateTime.Now;

bf.Serialize(file, progress);

file.Close();

}

/// <summary>

/// Loads the game progress from file.

/// </summary>

public void LoadGameProgress()

{

if (File.Exists(Application.persistentDataPath + gameProgressFile) == true)

{

BinaryFormatter bf = new BinaryFormatter();

FileStream file = File.Open(Application.persistentDataPath + gameProgressFile, FileMode.Open);

progress = (GameProgressData)bf.Deserialize(file);

file.Close();

}

}

/// <summary>

/// Saves the game configurations to file.

/// </summary>

public void SaveGameConfigs()

{

BinaryFormatter bf = new BinaryFormatter();

FileStream file = File.Create(Application.persistentDataPath + gameConfigsFile);

bf.Serialize(file, configs);

file.Close();

}

/// <summary>

/// Loads the game configurations from file.

/// </summary>

public void LoadGameConfigs()

{

if (File.Exists(Application.persistentDataPath + gameConfigsFile) == true)

{

BinaryFormatter bf = new BinaryFormatter();

FileStream file = File.Open(Application.persistentDataPath + gameConfigsFile, FileMode.Open);

configs = (GameConfigurations)bf.Deserialize(file);

file.Close();

}

}

}

using UnityEngine;

using UnityEngine.Events;

using System.Collections;

using System.Collections.Generic;

/// <summary>

/// My event type.

/// </summary>

[System.Serializable]

public class MyEvent : UnityEvent<GameObject, string>

{

}

/// <summary>

/// Message system.

/// </summary>

public class EventManager : MonoBehaviour

{

// Singleton

public static EventManager instance;

// Events list

private Dictionary<string, MyEvent> eventDictionary = new Dictionary<string, MyEvent>();

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

if (instance == null)

{

instance = this;

DontDestroyOnLoad(gameObject);

}

else if (instance != this)

{

Destroy(gameObject);

return;

}

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

if (instance == this)

{

instance = null;

}

}

/// <summary>

/// Start listening specified event.

/// </summary>

/// <param name="eventName">Event name.</param>

/// <param name="listener">Listener.</param>

public static void StartListening(string eventName, UnityAction<GameObject, string> listener)

{

if (instance == null)

{

instance = FindObjectOfType(typeof(EventManager)) as EventManager;

if (instance == null)

{

Debug.Log("Have no event manager on scene");

return;

}

}

MyEvent thisEvent = null;

if (instance.eventDictionary.TryGetValue(eventName, out thisEvent))

{

thisEvent.AddListener(listener);

}

else

{

thisEvent = new MyEvent();

thisEvent.AddListener(listener);

instance.eventDictionary.Add(eventName, thisEvent);

}

}

/// <summary>

/// Stop listening specified event.

/// </summary>

/// <param name="eventName">Event name.</param>

/// <param name="listener">Listener.</param>

public static void StopListening(string eventName, UnityAction<GameObject, string> listener)

{

if (instance == null)

{

return;

}

MyEvent thisEvent = null;

if (instance.eventDictionary.TryGetValue(eventName, out thisEvent))

{

thisEvent.RemoveListener(listener);

}

}

/// <summary>

/// Trigger specified event.

/// </summary>

/// <param name="eventName">Event name.</param>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

public static void TriggerEvent(string eventName, GameObject obj, string param)

{

if (instance == null)

{

return;

}

MyEvent thisEvent = null;

if (instance.eventDictionary.TryGetValue(eventName, out thisEvent))

{

thisEvent.Invoke(obj, param);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

/// <summary>

/// Level choose scene manager.

/// </summary>

public class LevelChoose : MonoBehaviour

{

// Scene to exit

public string exitSceneName;

// Visual displaing for number of levels

public Transform togglesFolder;

// Active toggle prefab

public Toggle activeTogglePrefab;

// Inactive toggle prefab

public Toggle inactiveTogglePrefab;

// Next level button

public Button nextLevelButton;

// Previous level button

public Button prevLevelButton;

// Folder for level visualisation

public Transform levelFolder;

// Choosen level

public GameObject currentLevel;

// All levels

public List<GameObject> levelsPrefabs = new List<GameObject>();

// Index of last allowed level for choosing

private int maxActiveLevelIdx;

// Index of current displayed level

private int currentDisplayedLevelIdx;

// List with active toggles

private List<Toggle> activeToggles = new List<Toggle>();

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("ButtonPressed", ButtonPressed);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("ButtonPressed", ButtonPressed);

}

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

maxActiveLevelIdx = -1;

Debug.Assert(currentLevel && togglesFolder && activeTogglePrefab && inactiveTogglePrefab && nextLevelButton && prevLevelButton && levelFolder, "Wrong initial settings");

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

int hitIdx = -1;

int levelsCount = DataManager.instance.progress.openedLevels.Count;

if (levelsCount > 0)

{

// Get name of last opened level from stored data

string openedLevelName = DataManager.instance.progress.openedLevels[levelsCount - 1];

int idx;

for (idx = 0; idx < levelsPrefabs.Count; ++idx)

{

// Try to find last opened level in levels list

if (levelsPrefabs[idx].name == openedLevelName)

{

hitIdx = idx;

break;

}

}

}

// Level found

if (hitIdx >= 0)

{

if (levelsPrefabs.Count > hitIdx + 1)

{

maxActiveLevelIdx = hitIdx + 1;

}

else

{

maxActiveLevelIdx = hitIdx;

}

}

// level does not found

else

{

if (levelsPrefabs.Count > 0)

{

maxActiveLevelIdx = 0;

}

else

{

Debug.LogError("Have no levels prefabs!");

}

}

if (maxActiveLevelIdx >= 0)

{

DisplayToggles();

DisplayLevel(maxActiveLevelIdx);

}

}

/// <summary>

/// Visual displaing for number of levels

/// </summary>

private void DisplayToggles()

{

foreach (Toggle toggle in togglesFolder.GetComponentsInChildren<Toggle>())

{

Destroy(toggle.gameObject);

}

int cnt;

for (cnt = 0; cnt < maxActiveLevelIdx + 1; cnt++)

{

GameObject toggle = Instantiate(activeTogglePrefab.gameObject, togglesFolder);

activeToggles.Add(toggle.GetComponent<Toggle>());

}

if (maxActiveLevelIdx < levelsPrefabs.Count - 1)

{

Instantiate(inactiveTogglePrefab.gameObject, togglesFolder);

}

}

/// <summary>

/// Displaies choosen level.

/// </summary>

/// <param name="levelIdx">Level index.</param>

private void DisplayLevel(int levelIdx)

{

Transform parentOfLevel = currentLevel.transform.parent;

Vector3 levelPosition = currentLevel.transform.position;

Quaternion levelRotation = currentLevel.transform.rotation;

Destroy(currentLevel);

currentLevel = Instantiate(levelsPrefabs[levelIdx], parentOfLevel);

currentLevel.name = levelsPrefabs[levelIdx].name;

currentLevel.transform.position = levelPosition;

currentLevel.transform.rotation = levelRotation;

currentDisplayedLevelIdx = levelIdx;

foreach (Toggle toggle in activeToggles)

{

toggle.isOn = false;

}

activeToggles[levelIdx].isOn = true;

UpdateButtonsVisible (levelIdx);

}

/// <summary>

/// Updates the buttons visible.

/// </summary>

/// <param name="levelIdx">Level index.</param>

private void UpdateButtonsVisible(int levelIdx)

{

prevLevelButton.interactable = levelIdx > 0 ? true : false;

nextLevelButton.interactable = levelIdx < maxActiveLevelIdx ? true : false;

}

/// <summary>

/// Displaies the next level.

/// </summary>

private void DisplayNextLevel()

{

if (currentDisplayedLevelIdx < maxActiveLevelIdx)

{

DisplayLevel(currentDisplayedLevelIdx + 1);

}

}

/// <summary>

/// Displaies the previous level.

/// </summary>

private void DisplayPrevLevel()

{

if (currentDisplayedLevelIdx > 0)

{

DisplayLevel (currentDisplayedLevelIdx - 1);

}

}

/// <summary>

/// Exit scene.

/// </summary>

private void Exit()

{

SceneManager.LoadScene(exitSceneName);

}

/// <summary>

/// Go to choosen level.

/// </summary>

private void GoToLevel()

{

SceneManager.LoadScene(currentLevel.name);

}

private void DisplayInfo()

{

SceneManager.LoadScene("LevelInfo");

}

/// <summary>

/// Buttons pressed handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void ButtonPressed(GameObject obj, string param)

{

switch (param)

{

case "Start":

GoToLevel();

break;

case "Info":

DisplayInfo();

break;

case "Exit":

Exit();

break;

case "Next":

DisplayNextLevel();

break;

case "Prev":

DisplayPrevLevel();

break;

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.SceneManagement;

public class LevelSwitcher : MonoBehaviour

{

public void BackButton()

{

SceneManager.LoadScene("LevelChoose");

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

/// <summary>

/// Main menu operate.

/// </summary>

public class MainMenu : MonoBehaviour

{

// Name of scene to start on click

public string startSceneName;

// Credits menu

public GameObject creditsMenu;

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("ButtonPressed", ButtonPressed);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("ButtonPressed", ButtonPressed);

}

void Awake()

{

Debug.Assert(creditsMenu, "Wrong initial settings");

}

/// <summary>

/// Buttons pressed handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void ButtonPressed(GameObject obj, string param)

{

switch (param)

{

case "Quit":

Application.Quit();

break;

case "Start":

SceneManager.LoadScene(startSceneName);

break;

case "OpenCredits":

creditsMenu.SetActive(true);

break;

case "CloseCredits":

creditsMenu.SetActive(false);

break;

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// This target can receive damage.

/// </summary>

public class DamageTaker : MonoBehaviour

{

// Start hitpoints

public int hitpoints = 1;

// Remaining hitpoints

[HideInInspector]

public int currentHitpoints;

// Damage visual effect duration

public float damageDisplayTime = 0.2f;

// Helth bar object

public Transform healthBar;

// SendMessage will trigger on damage taken

public bool isTrigger;

// Die sound effect

public AudioClip dieSfx;

// Image of this object

private SpriteRenderer sprite;

// Visualisation of hit or heal is in progress

private bool coroutineInProgress;

// Original width of health bar (full hp)

private float originHealthBarWidth;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

currentHitpoints = hitpoints;

sprite = GetComponentInChildren<SpriteRenderer>();

Debug.Assert(sprite && healthBar, "Wrong initial parameters");

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

originHealthBarWidth = healthBar.localScale.x;

}

/// <summary>

/// Take damage.

/// </summary>

/// <param name="damage">Damage.</param>

public void TakeDamage(int damage)

{

if (damage > 0)

{

if (this.enabled == true)

{

if (currentHitpoints > damage)

{

// Still alive

currentHitpoints -= damage;

UpdateHealthBar();

// If no coroutine now

if (coroutineInProgress == false)

{

// Damage visualisation

StartCoroutine(DisplayDamage());

}

if (isTrigger == true)

{

// Notify other components of this game object

SendMessage("OnDamage");

}

}

else

{

// Die

currentHitpoints = 0;

UpdateHealthBar();

Die();

}

}

}

else // damage < 0

{

// Healed

currentHitpoints = Mathf.Min(currentHitpoints - damage, hitpoints);

UpdateHealthBar();

}

}

/// <summary>

/// Updates the health bar width.

/// </summary>

private void UpdateHealthBar()

{

float healthBarWidth = originHealthBarWidth \* currentHitpoints / hitpoints;

healthBar.localScale = new Vector2(healthBarWidth, healthBar.localScale.y);

}

/// <summary>

/// Die this instance.

/// </summary>

public void Die()

{

EventManager.TriggerEvent("UnitKilled", gameObject, null);

StartCoroutine(DieCoroutine());

}

private IEnumerator DieCoroutine()

{

if (dieSfx != null && AudioManager.instance != null)

{

AudioManager.instance.PlayDie(dieSfx);

}

foreach (Collider2D col in GetComponentsInChildren<Collider2D>())

{

col.enabled = false;

}

GetComponent<AiBehavior>().enabled = false;

GetComponent<NavAgent>().enabled = false;

GetComponent<EffectControl>().enabled = false;

Animator anim = GetComponent<Animator>();

// If unit has animator

if (anim != null && anim.runtimeAnimatorController != null)

{

// Search for clip

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Die")

{

// Play animation

anim.SetTrigger("die");

yield return new WaitForSeconds(clip.length);

break;

}

}

}

Destroy(gameObject);

}

/// <summary>

/// Damage visualisation.

/// </summary>

/// <returns>The damage.</returns>

IEnumerator DisplayDamage()

{

coroutineInProgress = true;

Color originColor = sprite.color;

float counter;

// Set color to black and return to origin color over time

for (counter = 0f; counter < damageDisplayTime; counter += Time.fixedDeltaTime)

{

sprite.color = Color.Lerp(originColor, Color.black, Mathf.PingPong(counter, damageDisplayTime / 2f));

yield return new WaitForFixedUpdate();

}

sprite.color = originColor;

coroutineInProgress = false;

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

EventManager.TriggerEvent("UnitDie", gameObject, null);

StopAllCoroutines();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Price of tower or unit.

/// </summary>

public class Price : MonoBehaviour

{

// Price in gold

public int price;

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Operate sprite order depend on y position.

/// </summary>

public class SpriteSorting : MonoBehaviour

{

// Static will no change order on update, only on start

public bool isStatic;

// Multiplier for accuracy inreasing

public float rangeFactor = 100f;

// Sprites list for this object in clildren

private Dictionary<SpriteRenderer, int> sprites = new Dictionary<SpriteRenderer, int>();

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

foreach (SpriteRenderer sprite in GetComponentsInChildren<SpriteRenderer>())

{

sprites.Add(sprite, sprite.sortingOrder);

}

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

UpdateSortingOrder();

}

/// <summary>

/// Update this instance.

/// </summary>

void Update()

{

if (isStatic == false)

{

UpdateSortingOrder();

}

}

/// <summary>

/// Update sprites sorting order.

/// </summary>

private void UpdateSortingOrder()

{

foreach (KeyValuePair<SpriteRenderer, int> sprite in sprites)

{

sprite.Key.sortingOrder = sprite.Value - (int)(transform.position.y \* rangeFactor);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// Unit info, displayed on click.

/// </summary>

public class UnitInfo : MonoBehaviour

{

public string unitName;

public Sprite primaryIcon;

public string primaryText;

public Sprite secondaryIcon;

public string secondaryText;

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.SceneManagement;

/// <summary>

/// Level control script.

/// </summary>

public class LevelManager : MonoBehaviour

{

// UI scene. Load on level start

public string levelUiSceneName;

// Gold amount for this level

public int goldAmount = 50;

// How many times enemies can reach capture point before defeat

public int defeatAttempts = 1;

// List with allowed randomly generated enemy for this level

public List<GameObject> allowedEnemies = new List<GameObject>();

// List with allowed towers for this level

public List<GameObject> allowedTowers = new List<GameObject>();

// List with allowed spells for this level

public List<GameObject> allowedSpells = new List<GameObject>();

// User interface manager

private UiManager uiManager;

// Nymbers of enemy spawners in this level

private int spawnNumbers;

// Current loose counter

private int beforeLooseCounter;

// Victory or defeat condition already triggered

private bool triggered = false;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

// Load UI scene

SceneManager.LoadScene(levelUiSceneName, LoadSceneMode.Additive);

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

uiManager = FindObjectOfType<UiManager>();

SpawnPoint[] spawnPoints = FindObjectsOfType<SpawnPoint>();

spawnNumbers = spawnPoints.Length;

if (spawnNumbers <= 0)

{

Debug.LogError("Have no spawners");

}

// Set random enemies list for each spawner

foreach (SpawnPoint spawnPoint in spawnPoints)

{

spawnPoint.randomEnemiesList = allowedEnemies;

}

Debug.Assert(uiManager, "Wrong initial parameters");

// Set gold amount for this level

uiManager.SetGold(goldAmount);

beforeLooseCounter = defeatAttempts;

uiManager.SetDefeatAttempts(beforeLooseCounter);

}

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("Captured", Captured);

EventManager.StartListening("AllEnemiesAreDead", AllEnemiesAreDead);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("Captured", Captured);

EventManager.StopListening("AllEnemiesAreDead", AllEnemiesAreDead);

}

/// <summary>

/// Enemy reached capture point.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void Captured(GameObject obj, string param)

{

if (beforeLooseCounter > 0)

{

beforeLooseCounter--;

uiManager.SetDefeatAttempts(beforeLooseCounter);

if (beforeLooseCounter <= 0)

{

triggered = true;

// Defeat

EventManager.TriggerEvent("Defeat", null, null);

}

}

}

/// <summary>

/// All enemies are dead.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void AllEnemiesAreDead(GameObject obj, string param)

{

spawnNumbers--;

// Enemies dead at all spawners

if (spawnNumbers <= 0)

{

// Check if loose condition was not triggered before

if (triggered == false)

{

// Victory

EventManager.TriggerEvent("Victory", null, null);

}

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Enemy waves timings.

/// </summary>

[ExecuteInEditMode]

public class WavesInfo : MonoBehaviour

{

// TO between waves

public List<float> wavesTimeouts = new List<float>();

// Execute only in edit mode

#if UNITY\_EDITOR

// TO between waves by default

private float defaultWaveTimeout = 30f;

// List with active spawners in level

private SpawnPoint[] spawners;

/// <summary>

/// On editor update.

/// </summary>

public void Update()

{

spawners = FindObjectsOfType<SpawnPoint>();

int wavesCount = 0;

// Get the max number of waves from spawners

foreach (SpawnPoint spawner in spawners)

{

if (spawner.waves.Count > wavesCount)

{

wavesCount = spawner.waves.Count;

}

}

// Display actual list with waves timeouts

if (wavesTimeouts.Count < wavesCount)

{

int i;

for (i = wavesTimeouts.Count; i < wavesCount; ++i)

{

wavesTimeouts.Add (defaultWaveTimeout);

}

}

else if (wavesTimeouts.Count > wavesCount)

{

wavesTimeouts.RemoveRange (wavesCount, wavesTimeouts.Count - wavesCount);

}

}

#endif

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using System.Linq;

/// <summary>

/// Enemy spawner.

/// </summary>

public class SpawnPoint : MonoBehaviour

{

/// <summary>

/// Enemy wave structure.

/// </summary>

[System.Serializable]

public class Wave

{

// Delay before wave run

public float delayBeforeWave;

// List of enemies in this wave

public List<GameObject> enemies = new List<GameObject>();

}

// Enemies will have different speed in specified interval

public float speedRandomizer = 0.2f;

// Delay between enemies spawn in wave

public float unitSpawnDelay = 1.5f;

// Waves list for this spawner

public List<Wave> waves;

// Endless enemies wave mode for this spawn poin

public bool endlessWave = false;

// This list is used for random enemy spawn

[HideInInspector]

public List<GameObject> randomEnemiesList = new List<GameObject>();

// Enemies will move along this pathway

private Pathway path;

// Delay counter

private float counter;

// Buffer with active spawned enemies

private List<GameObject> activeEnemies = new List<GameObject>();

// All enemies were spawned

private bool finished = false;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake ()

{

path = GetComponentInParent<Pathway>();

Debug.Assert(path != null, "Wrong initial parameters");

}

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("UnitDie", UnitDie);

EventManager.StartListening("WaveStart", WaveStart);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("UnitDie", UnitDie);

EventManager.StopListening("WaveStart", WaveStart);

}

/// <summary>

/// Update this instance.

/// </summary>

void Update()

{

// If all spawned enemies are dead

if ((finished == true) && (activeEnemies.Count <= 0))

{

EventManager.TriggerEvent("AllEnemiesAreDead", null, null);

gameObject.SetActive(false);

}

}

/// <summary>

/// Runs the wave.

/// </summary>

/// <returns>The wave.</returns>

private IEnumerator RunWave(int waveIdx)

{

if (waves.Count > waveIdx)

{

yield return new WaitForSeconds(waves[waveIdx].delayBeforeWave);

while (endlessWave == true)

{

GameObject prefab = randomEnemiesList[Random.Range (0, randomEnemiesList.Count)];

// Create enemy

GameObject newEnemy = Instantiate(prefab, transform.position, transform.rotation);

newEnemy.name = prefab.name;

// Set pathway

newEnemy.GetComponent<AiStatePatrol>().path = path;

NavAgent agent = newEnemy.GetComponent<NavAgent>();

// Set speed offset

agent.speed = Random.Range(agent.speed \* (1f - speedRandomizer), agent.speed \* (1f + speedRandomizer));

// Add enemy to list

activeEnemies.Add(newEnemy);

// Wait for delay before next enemy run

yield return new WaitForSeconds(unitSpawnDelay);

}

foreach (GameObject enemy in waves[waveIdx].enemies)

{

GameObject prefab = null;

prefab = enemy;

// If enemy prefab not specified - spawn random enemy

if (prefab == null && randomEnemiesList.Count > 0)

{

prefab = randomEnemiesList[Random.Range (0, randomEnemiesList.Count)];

}

if (prefab == null)

{

Debug.LogError("Have no enemy prefab. Please specify enemies in Level Manager or in Spawn Point");

}

// Create enemy

GameObject newEnemy = Instantiate(prefab, transform.position, transform.rotation);

newEnemy.name = prefab.name;

// Set pathway

newEnemy.GetComponent<AiStatePatrol>().path = path;

NavAgent agent = newEnemy.GetComponent<NavAgent>();

// Set speed offset

agent.speed = Random.Range(agent.speed \* (1f - speedRandomizer), agent.speed \* (1f + speedRandomizer));

// Add enemy to list

activeEnemies.Add(newEnemy);

// Wait for delay before next enemy run

yield return new WaitForSeconds(unitSpawnDelay);

}

if (waveIdx + 1 == waves.Count)

{

finished = true;

}

}

}

/// <summary>

/// On unit die.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UnitDie(GameObject obj, string param)

{

// If this is active enemy

if (activeEnemies.Contains(obj) == true)

{

// Remove it from buffer

activeEnemies.Remove(obj);

}

}

// Wave start event received

private void WaveStart(GameObject obj, string param)

{

int waveIdx;

int.TryParse(param, out waveIdx);

StartCoroutine("RunWave", waveIdx);

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

StopAllCoroutines();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Pathway for enemy moving.

/// </summary>

[ExecuteInEditMode]

public class Pathway : MonoBehaviour

{

#if UNITY\_EDITOR

/// <summary>

/// Update this instance.

/// </summary>

void Update()

{

Waypoint[] waypoints = GetComponentsInChildren<Waypoint>();

if (waypoints.Length > 1)

{

int idx;

for (idx = 1; idx < waypoints.Length; idx++)

{

// Draw blue lines along pathway in edit mode

Debug.DrawLine(waypoints[idx - 1].transform.position, waypoints[idx].transform.position, new Color(0.7f, 0f, 0f));

}

}

}

#endif

/// <summary>

/// Gets the nearest waypoint for specified position.

/// </summary>

/// <returns>The nearest waypoint.</returns>

/// <param name="position">Position.</param>

public Waypoint GetNearestWaypoint(Vector3 position)

{

float minDistance = float.MaxValue;

Waypoint nearestWaypoint = null;

foreach (Waypoint waypoint in GetComponentsInChildren<Waypoint>())

{

// Calculate distance to waypoint

Vector3 vect = position - waypoint.transform.position;

float distance = vect.magnitude;

if (distance < minDistance)

{

minDistance = distance;

nearestWaypoint = waypoint;

}

}

for (;;)

{

float waypointPathDistance = GetPathDistance(nearestWaypoint);

Waypoint nextWaypoint = GetNextWaypoint(nearestWaypoint, false);

if (nextWaypoint != null)

{

float myPathDistance = GetPathDistance(nextWaypoint) + (nextWaypoint.transform.position - position).magnitude;

if (waypointPathDistance <= myPathDistance)

{

break;

}

else

{

nearestWaypoint = nextWaypoint;

}

}

else

{

break;

}

}

return nearestWaypoint;

}

/// <summary>

/// Gets the next waypoint on this pathway.

/// </summary>

/// <returns>The next waypoint.</returns>

/// <param name="currentWaypoint">Current waypoint.</param>

/// <param name="loop">If set to <c>true</c> loop.</param>

public Waypoint GetNextWaypoint(Waypoint currentWaypoint, bool loop)

{

Waypoint res = null;

int idx = currentWaypoint.transform.GetSiblingIndex();

if (idx < (transform.childCount - 1))

{

idx += 1;

}

else

{

idx = 0;

}

if (!(loop == false && idx == 0))

{

res = transform.GetChild(idx).GetComponent<Waypoint>();

}

return res;

}

/// <summary>

/// Gets the previous waypoint on this pathway.

/// </summary>

/// <returns>The previous waypoint.</returns>

/// <param name="currentWaypoint">Current waypoint.</param>

/// <param name="loop">If set to <c>true</c> loop.</param>

public Waypoint GetPreviousWaypoint(Waypoint currentWaypoint, bool loop)

{

Waypoint res = null;

int idx = currentWaypoint.transform.GetSiblingIndex();

if (idx > 0)

{

idx -= 1;

}

else

{

idx = transform.childCount - 1;

}

if (!(loop == false && idx == transform.childCount - 1))

{

res = transform.GetChild(idx).GetComponent<Waypoint>();

}

return res;

}

/// <summary>

/// Gets the remaining path distance from specified waypoint.

/// </summary>

/// <returns>The path distance.</returns>

/// <param name="fromWaypoint">From waypoint.</param>

public float GetPathDistance(Waypoint fromWaypoint)

{

Waypoint[] waypoints = GetComponentsInChildren<Waypoint>();

bool hitted = false;

float pathDistance = 0f;

int idx;

// Calculate remaining path

for (idx = 0; idx < waypoints.Length; ++idx)

{

if (hitted == true)

{

// Add distance between waypoint to result

Vector2 distance = waypoints[idx].transform.position - waypoints[idx - 1].transform.position;

pathDistance += distance.magnitude;

}

if (waypoints[idx] == fromWaypoint)

{

hitted = true;

}

}

return pathDistance;

}

/// <summary>

/// Gets the offset position on pathway.

/// </summary>

/// <returns>The offset position.</returns>

/// <param name="nextWaypoint">Next waypoint. Will be updated after offset</param>

/// <param name="currentPosition">Current position.</param>

/// <param name="offsetDistance">Offset distance.</param>

public Vector2 GetOffsetPosition(ref Waypoint nextWaypoint, Vector2 currentPosition, float offsetDistance)

{

Vector2 res = currentPosition;

if (offsetDistance >= 0f) // Forward offset

{

float remainingDistance = offsetDistance;

Vector2 lastPosition = currentPosition;

Waypoint waypoint = nextWaypoint;

Vector2 deltaVector = Vector2.zero;

// Calculate waypoint nearest to offset position

for (;;)

{

deltaVector = (Vector2)waypoint.transform.position - lastPosition;

float deltaDistance = deltaVector.magnitude;

if (remainingDistance > deltaDistance)

{

remainingDistance -= deltaDistance;

lastPosition = waypoint.transform.position;

waypoint = GetNextWaypoint(waypoint, false);

if (waypoint == null)

{

remainingDistance = 0f;

break;

}

else

{

nextWaypoint = waypoint;

}

}

else

{

break;

}

}

// Calculate result position

res = lastPosition + deltaVector.normalized \* remainingDistance;

}

else // Back offset

{

float remainingDistance = -offsetDistance;

Vector2 lastPosition = currentPosition;

Waypoint waypoint = GetPreviousWaypoint(nextWaypoint, false);

if (waypoint == null)

{

return res;

}

Vector2 deltaVector = Vector2.zero;

// Calculate waypoint nearest to offset position

for (;;)

{

deltaVector = (Vector2)waypoint.transform.position - lastPosition;

float deltaDistance = deltaVector.magnitude;

if (remainingDistance > deltaDistance)

{

nextWaypoint = waypoint;

remainingDistance -= deltaDistance;

lastPosition = waypoint.transform.position;

waypoint = GetPreviousWaypoint(waypoint, false);

if (waypoint == null)

{

remainingDistance = 0f;

break;

}

}

else

{

break;

}

}

// Calculate result position

res = lastPosition + deltaVector.normalized \* remainingDistance;

}

return res;

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Allows tower to spawn new obects with cooldown.

/// </summary>

public class DefendersSpawner : MonoBehaviour

{

// Cooldown for between spawns

public float cooldown = 10f;

// Max number of spawned obects in buffer

public int maxNum = 2;

// Spawned object prefab

public GameObject prefab;

// Position for spawning

public Transform spawnPoint;

[HideInInspector]

// Defend points for this tower

public DefendPoint defPoint;

// Counter for cooldown calculation

private float cooldownCounter;

// Animator of this instance

private Animator anim;

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

anim = GetComponent<Animator>();

Debug.Assert(spawnPoint, "Wrong initial settings");

BuildingPlace buildingPlace = GetComponentInParent<BuildingPlace>();

defPoint = buildingPlace.GetComponentInChildren<DefendPoint>();

cooldownCounter = cooldown;

// Upgrade all existing defenders on tower build

Dictionary<GameObject, Transform> oldDefenders = new Dictionary<GameObject, Transform>();

foreach (KeyValuePair<GameObject, Transform> pair in defPoint.activeDefenders)

{

oldDefenders.Add(pair.Key, pair.Value);

}

defPoint.activeDefenders.Clear();

foreach (KeyValuePair<GameObject, Transform> pair in oldDefenders)

{

// Spawn new defender in the same place

Spawn(pair.Key.transform, pair.Value);

}

// Destroy old defenders

foreach (KeyValuePair<GameObject, Transform> pair in oldDefenders)

{

Destroy(pair.Key);

}

}

/// <summary>

/// Update this instance.

/// </summary>

void FixedUpdate()

{

cooldownCounter += Time.fixedDeltaTime;

if (cooldownCounter >= cooldown)

{

// Try to spawn new object on cooldown

if (TryToSpawn() == true)

{

cooldownCounter = 0f;

}

else

{

cooldownCounter = cooldown;

}

}

}

/// <summary>

/// Try to spawn new object.

/// </summary>

/// <returns><c>true</c>, if to spawn was tryed, <c>false</c> otherwise.</returns>

private bool TryToSpawn()

{

bool res = false;

// If spawned objects number less then max allowed number

if ((prefab != null) && (defPoint.activeDefenders.Count < maxNum))

{

Transform destination = defPoint.GetFreeDefendPosition();

// If there are free defend position

if (destination != null)

{

// Spawn new defender

Spawn(spawnPoint, destination);

res = true;

}

}

return res;

}

/// <summary>

/// Spawn in the specified position and destination.

/// </summary>

/// <param name="position">Position.</param>

/// <param name="destination">Destination.</param>

private void Spawn(Transform position, Transform destination)

{

// Create new obect

GameObject obj = Instantiate<GameObject>(prefab, position.position, position.rotation);

obj.name = prefab.name;

// Set destination position

obj.GetComponent<AiStateMove>().destination = destination;

// Add spawned object to list

defPoint.activeDefenders.Add(obj, destination);

// Play animation

if (anim != null && anim.runtimeAnimatorController != null)

{

foreach (AnimationClip clip in anim.runtimeAnimatorController.animationClips)

{

if (clip.name == "Spawn")

{

anim.SetTrigger("spawn");

break;

}

}

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Tower building and operation.

/// </summary>

public class Tower : MonoBehaviour

{

// Prefab for actions tree

public GameObject actions;

// Visualisation of attack or defend range for this tower

public GameObject range;

// User interface manager

private UiManager uiManager;

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("GamePaused", GamePaused);

EventManager.StartListening("UserClick", UserClick);

EventManager.StartListening("UserUiClick", UserClick);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("GamePaused", GamePaused);

EventManager.StopListening("UserClick", UserClick);

EventManager.StopListening("UserUiClick", UserClick);

}

/// <summary>

/// Atart this instance.

/// </summary>

void Start()

{

uiManager = FindObjectOfType<UiManager>();

Debug.Assert(uiManager && actions, "Wrong initial parameters");

CloseActions();

}

/// <summary>

/// Opens the actions tree.

/// </summary>

private void OpenActions()

{

actions.SetActive(true);

}

/// <summary>

/// Closes the actions tree.

/// </summary>

private void CloseActions()

{

if (actions.activeSelf == true)

{

actions.SetActive(false);

}

}

/// <summary>

/// Builds the tower.

/// </summary>

/// <param name="towerPrefab">Tower prefab.</param>

public void BuildTower(GameObject towerPrefab)

{

// Close active actions tree

CloseActions();

Price price = towerPrefab.GetComponent<Price>();

// If anough gold

if (uiManager.SpendGold(price.price) == true)

{

// Create new tower and place it on same position

GameObject newTower = Instantiate<GameObject>(towerPrefab, transform.parent);

newTower.name = towerPrefab.name;

newTower.transform.position = transform.position;

newTower.transform.rotation = transform.rotation;

// Destroy old tower

Destroy(gameObject);

EventManager.TriggerEvent("TowerBuild", newTower, null);

}

}

/// <summary>

/// Sells the tower with half of price.

/// </summary>

/// <param name="emptyPlacePrefab">Empty place prefab.</param>

public void SellTower(GameObject emptyPlacePrefab)

{

CloseActions();

DefendersSpawner defendersSpawner = GetComponent<DefendersSpawner>();

// Destroy defenders on tower sell

if (defendersSpawner != null)

{

foreach (KeyValuePair<GameObject, Transform> pair in defendersSpawner.defPoint.activeDefenders)

{

Destroy(pair.Key);

}

}

Price price = GetComponent<Price>();

uiManager.AddGold(price.price / 2);

// Place building place

GameObject newTower = Instantiate<GameObject>(emptyPlacePrefab, transform.parent);

newTower.name = emptyPlacePrefab.name;

newTower.transform.position = transform.position;

newTower.transform.rotation = transform.rotation;

// Destroy old tower

Destroy(gameObject);

EventManager.TriggerEvent("TowerSell", null, null);

}

/// <summary>

/// Disable tower raycast and close building tree on game pause.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void GamePaused(GameObject obj, string param)

{

if (param == bool.TrueString) // Paused

{

CloseActions();

}

}

/// <summary>

/// On user click.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UserClick(GameObject obj, string param)

{

if (obj == gameObject) // This tower is clicked

{

// Show range

ShowRange(true);

if (actions.activeSelf == false)

{

// Open building tree if it is not

OpenActions();

}

}

else // Other click

{

// Hide range

ShowRange(false);

// Close active building tree

CloseActions();

}

}

/// <summary>

/// Display tower's attack or defend range.

/// </summary>

/// <param name="condition">If set to <c>true</c> condition.</param>

public void ShowRange(bool condition)

{

if (range != null)

{

range.SetActive(condition);

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

/// <summary>

/// Basic class for tower action.

/// </summary>

public class TowerAction : MonoBehaviour

{

// Icon for enabled state

public GameObject enabledIcon;

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("UserUiClick", UserUiClick);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("UserUiClick", UserUiClick);

}

/// <summary>

/// On user UI click.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UserUiClick(GameObject obj, string param)

{

// If clicked on this icon

if (obj == gameObject)

{

if (enabledIcon.activeSelf == true)

{

Clicked();

}

}

}

/// <summary>

/// Clicked this instance.

/// </summary>

protected virtual void Clicked()

{

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// Build the tower.

/// </summary>

public class TowerActionBuild : TowerAction

{

// Tower prefab for this icon

public GameObject towerPrefab;

// Icon for disabled state

public GameObject disabledIcon;

// Icon for blocked state while player has not anough gold

public GameObject blockedIcon;

// Text field for tower price

private Text priceText;

// Price of tower in gold

private int price = 0;

// Level manger has a list with allowed tower upgrades for this level.

private LevelManager levelManager;

// User interface manager allows to check current gold amount

private UiManager uiManager;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

priceText = GetComponentInChildren<Text>();

levelManager = FindObjectOfType<LevelManager>();

uiManager = FindObjectOfType<UiManager>();

Debug.Assert(priceText && towerPrefab && enabledIcon && disabledIcon && levelManager && uiManager, "Wrong initial parameters");

// Display tower price

price = towerPrefab.GetComponent<Price>().price;

priceText.text = price.ToString();

if (levelManager.allowedTowers.Contains(towerPrefab) == true)

{

enabledIcon.SetActive(true);

disabledIcon.SetActive(false);

}

else

{

enabledIcon.SetActive(false);

disabledIcon.SetActive(true);

}

}

/// <summary>

/// Update this instance.

/// </summary>

void Update()

{

// Mask build icon wich blocking icon if player has not anough gold

if (enabledIcon == true && blockedIcon != null)

{

if (uiManager.GetGold() >= price)

{

blockedIcon.SetActive(false);

}

else

{

blockedIcon.SetActive(true);

}

}

}

/// <summary>

/// Clicked this instance.

/// </summary>

protected override void Clicked()

{

// If player has anough gold

if (blockedIcon == null || blockedIcon.activeSelf == false)

{

// Build the tower

Tower tower = GetComponentInParent<Tower>();

if (tower != null)

{

tower.BuildTower(towerPrefab);

}

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// Button handler.

/// </summary>

public class ButtonHandler : MonoBehaviour

{

public AudioClip audioClip;

/// <summary>

/// Buttons pressed.

/// </summary>

/// <param name="buttonName">Button name.</param>

public void ButtonPressed(string buttonName)

{

StartCoroutine(PressedCoroutine(buttonName));

}

/// <summary>

/// Presseds the coroutine.

/// </summary>

/// <returns>The coroutine.</returns>

/// <param name="buttonName">Button name.</param>

private IEnumerator PressedCoroutine(string buttonName)

{

// Play sound effect

if (audioClip != null && AudioManager.instance != null)

{

Button button = GetComponent<Button>();

button.interactable = false;

AudioManager.instance.PlaySound(audioClip);

// Wayt for sound effect end

yield return new WaitForSecondsRealtime(audioClip.length);

button.interactable = true;

}

// Send global event about button preesing

EventManager.TriggerEvent("ButtonPressed", gameObject, buttonName);

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

StopAllCoroutines();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// Show unit info on special sheet.

/// </summary>

public class ShowInfo : MonoBehaviour

{

// Name of unit

public Text unitName;

// Primary icon for displaing

public Image primaryIcon;

// Primary text for displaing

public Text primaryText;

// Secondary icon for displaing

public Image secondaryIcon;

// Secondary text for displaing

public Text secondaryText;

// Hitpoints icon for displaying

public Sprite hitpointsIcon;

// Melee attack icon for displaying

public Sprite meleeAttackIcon;

// Ranged attack icon for displaying

public Sprite rangedAttackIcon;

// Defenders number icon for displaying

public Sprite defendersNumberIcon;

// Cooldown icon for displaying

public Sprite cooldownIcon;

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

EventManager.StopListening("UserClick", UserClick);

}

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

Debug.Assert(unitName && primaryIcon && primaryText && secondaryIcon && secondaryText, "Wrong intial settings");

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

EventManager.StartListening("UserClick", UserClick);

HideUnitInfo();

}

/// <summary>

/// Shows the unit info.

/// </summary>

/// <param name="info">Info.</param>

public void ShowUnitInfo(UnitInfo info, GameObject obj)

{

if (info.unitName != "")

{

unitName.text = info.unitName;

}

else

{

unitName.text = obj.name;

}

if (info.primaryIcon != null || info.secondaryIcon != null || info.primaryText != "" || info.secondaryText != "")

{

primaryText.text = info.primaryText;

secondaryText.text = info.secondaryText;

if (info.primaryIcon != null)

{

primaryIcon.sprite = info.primaryIcon;

primaryIcon.gameObject.SetActive(true);

}

if (info.secondaryIcon != null)

{

secondaryIcon.sprite = info.secondaryIcon;

secondaryIcon.gameObject.SetActive(true);

}

}

else

{

DamageTaker damageTaker = obj.GetComponentInChildren<DamageTaker>();

Attack attack = obj.GetComponentInChildren<Attack>();

DefendersSpawner spawner = obj.GetComponentInChildren<DefendersSpawner>();

// Automaticaly set primary icon and text

if (damageTaker != null)

{

primaryText.text = damageTaker.hitpoints.ToString();

primaryIcon.sprite = hitpointsIcon;

primaryIcon.gameObject.SetActive(true);

}

else

{

if (attack != null)

{

if (attack != null)

{

primaryText.text = attack.cooldown.ToString();

primaryIcon.sprite = cooldownIcon;

primaryIcon.gameObject.SetActive(true);

}

}

else if (spawner != null)

{

primaryText.text = spawner.cooldown.ToString();

primaryIcon.sprite = cooldownIcon;

primaryIcon.gameObject.SetActive(true);

}

}

if (attack != null)

{

secondaryText.text = attack.damage.ToString();

if (attack is AttackMelee)

{

secondaryIcon.sprite = meleeAttackIcon;

}

else if (attack is AttackRanged)

{

secondaryIcon.sprite = rangedAttackIcon;

}

secondaryIcon.gameObject.SetActive(true);

}

else

{

if (spawner != null)

{

secondaryText.text = spawner.maxNum.ToString();

secondaryIcon.sprite = defendersNumberIcon;

secondaryIcon.gameObject.SetActive(true);

}

}

}

gameObject.SetActive(true);

}

/// <summary>

/// Hides the unit info.

/// </summary>

public void HideUnitInfo()

{

unitName.text = primaryText.text = secondaryText.text = "";

primaryIcon.gameObject.SetActive(false);

secondaryIcon.gameObject.SetActive(false);

gameObject.SetActive(false);

}

/// <summary>

/// User click handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UserClick(GameObject obj, string param)

{

HideUnitInfo();

if (obj != null)

{

// Cliced object has info for displaing

UnitInfo unitInfo = obj.GetComponentInChildren<UnitInfo>();

if (unitInfo != null)

{

ShowUnitInfo(unitInfo, obj);

}

}

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

using UnityEngine.EventSystems;

using UnityEngine.SceneManagement;

/// <summary>

/// User interface and events manager.

/// </summary>

public class UiManager : MonoBehaviour

{

// This scene will loaded after whis level exit

public string exitSceneName;

// Start screen canvas

public GameObject startScreen;

// Pause menu canvas

public GameObject pauseMenu;

// Defeat menu canvas

public GameObject defeatMenu;

// Victory menu canvas

public GameObject victoryMenu;

// Level interface

public GameObject levelUI;

// Avaliable gold amount

public Text goldAmount;

// Capture attempts before defeat

public Text defeatAttempts;

// Victory and defeat menu display delay

public float menuDisplayDelay = 1f;

// Is game paused?

private bool paused;

// Camera is dragging now

private bool cameraIsDragged;

// Origin point of camera dragging start

private Vector3 dragOrigin = Vector3.zero;

// Camera control component

private CameraControl cameraControl;

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

cameraControl = FindObjectOfType<CameraControl>();

Debug.Assert(cameraControl && startScreen && pauseMenu && defeatMenu && victoryMenu && levelUI && defeatAttempts && goldAmount, "Wrong initial parameters");

}

/// <summary>

/// Raises the enable event.

/// </summary>

void OnEnable()

{

EventManager.StartListening("UnitKilled", UnitKilled);

EventManager.StartListening("ButtonPressed", ButtonPressed);

EventManager.StartListening("Defeat", Defeat);

EventManager.StartListening("Victory", Victory);

}

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

EventManager.StopListening("UnitKilled", UnitKilled);

EventManager.StopListening("ButtonPressed", ButtonPressed);

EventManager.StopListening("Defeat", Defeat);

EventManager.StopListening("Victory", Victory);

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

PauseGame(true);

}

/// <summary>

/// Update this instance.

/// </summary>

void Update()

{

if (paused == false)

{

// User press mouse button

if (Input.GetMouseButtonDown(0) == true)

{

// Check if pointer over UI components

GameObject hittedObj = null;

PointerEventData pointerData = new PointerEventData(EventSystem.current);

pointerData.position = Input.mousePosition;

List<RaycastResult> results = new List<RaycastResult>();

EventSystem.current.RaycastAll(pointerData, results);

if (results.Count > 0) // UI components on pointer

{

// Search for Action Icon hit in results

foreach (RaycastResult res in results)

{

if (res.gameObject.CompareTag("ActionIcon"))

{

hittedObj = res.gameObject;

break;

}

}

// Send message with user click data on UI component

EventManager.TriggerEvent("UserUiClick", hittedObj, null);

}

else // No UI components on pointer

{

// Check if pointer over colliders

RaycastHit2D[] hits = Physics2D.RaycastAll(Camera.main.ScreenToWorldPoint(Input.mousePosition), Camera.main.transform.forward);

foreach (RaycastHit2D hit in hits)

{

// If this object has unit info

if (hit.collider.CompareTag("UnitInfo"))

{

Tower tower = hit.collider.GetComponentInParent<Tower>();

if (tower != null)

{

hittedObj = tower.gameObject;

break;

}

AiBehavior aiBehavior = hit.collider.GetComponentInParent<AiBehavior>();

if (aiBehavior != null)

{

hittedObj = aiBehavior.gameObject;

break;

}

hittedObj = hit.collider.gameObject;

break;

}

}

// Send message with user click data on game space

EventManager.TriggerEvent("UserClick", hittedObj, null);

}

// If there is no hitted object - start camera drag

if (hittedObj == null)

{

cameraIsDragged = true;

dragOrigin = Input.mousePosition;

}

}

if (Input.GetMouseButtonUp(0) == true)

{

// Stop drag camera on mouse release

cameraIsDragged = false;

}

if (cameraIsDragged == true)

{

Vector3 pos = Camera.main.ScreenToViewportPoint(Input.mousePosition - dragOrigin);

// Camera dragging (inverted)

cameraControl.MoveX(-pos.x);

cameraControl.MoveY(-pos.y);

}

}

}

/// <summary>

/// Stop current scene and load new scene

/// </summary>

/// <param name="sceneName">Scene name.</param>

private void LoadScene(string sceneName)

{

EventManager.TriggerEvent("SceneQuit", null, null);

SceneManager.LoadScene(sceneName);

}

/// <summary>

/// Resumes the game.

/// </summary>

private void ResumeGame()

{

GoToLevel();

PauseGame(false);

}

/// <summary>

/// Gos to main menu.

/// </summary>

private void ExitFromLevel()

{

LoadScene(exitSceneName);

}

/// <summary>

/// Closes all UI canvases.

/// </summary>

private void CloseAllUI()

{

startScreen.SetActive (false);

pauseMenu.SetActive(false);

defeatMenu.SetActive(false);

victoryMenu.SetActive(false);

}

/// <summary>

/// Pauses the game.

/// </summary>

/// <param name="pause">If set to <c>true</c> pause.</param>

private void PauseGame(bool pause)

{

paused = pause;

// Stop the time on pause

Time.timeScale = pause ? 0f : 1f;

EventManager.TriggerEvent("GamePaused", null, pause.ToString());

}

/// <summary>

/// Gos to pause menu.

/// </summary>

private void GoToPauseMenu()

{

PauseGame(true);

CloseAllUI();

pauseMenu.SetActive(true);

}

/// <summary>

/// Gos to level.

/// </summary>

private void GoToLevel()

{

CloseAllUI();

levelUI.SetActive(true);

PauseGame(false);

}

/// <summary>

/// Gos to defeat menu.

/// </summary>

private void Defeat(GameObject obj, string param)

{

StartCoroutine("DefeatCoroutine");

}

/// <summary>

/// Display defeat menu after delay.

/// </summary>

/// <returns>The coroutine.</returns>

private IEnumerator DefeatCoroutine()

{

yield return new WaitForSeconds(menuDisplayDelay);

PauseGame(true);

CloseAllUI();

defeatMenu.SetActive(true);

}

/// <summary>

/// Gos to victory menu.

/// </summary>

private void Victory(GameObject obj, string param)

{

StartCoroutine("VictoryCoroutine");

}

/// <summary>

/// Display victory menu after delay.

/// </summary>

/// <returns>The coroutine.</returns>

private IEnumerator VictoryCoroutine()

{

yield return new WaitForSeconds(menuDisplayDelay);

PauseGame(true);

CloseAllUI();

// --- Game progress autosaving ---

// Get the name of completed level

DataManager.instance.progress.lastCompetedLevel = SceneManager.GetActiveScene().name;

// Check if this level have no completed before

bool hit = false;

foreach (string level in DataManager.instance.progress.openedLevels)

{

if (level == SceneManager.GetActiveScene().name)

{

hit = true;

break;

}

}

if (hit == false)

{

DataManager.instance.progress.openedLevels.Add(SceneManager.GetActiveScene().name);

}

// Save game progress

DataManager.instance.SaveGameProgress();

victoryMenu.SetActive(true);

}

/// <summary>

/// Restarts current level.

/// </summary>

private void RestartLevel()

{

LoadScene(SceneManager.GetActiveScene().name);

}

/// <summary>

/// Gets current gold amount.

/// </summary>

/// <returns>The gold.</returns>

public int GetGold()

{

int gold;

int.TryParse(goldAmount.text, out gold);

return gold;

}

/// <summary>

/// Sets gold amount.

/// </summary>

/// <param name="gold">Gold.</param>

public void SetGold(int gold)

{

goldAmount.text = gold.ToString();

}

/// <summary>

/// Adds the gold.

/// </summary>

/// <param name="gold">Gold.</param>

public void AddGold(int gold)

{

SetGold(GetGold() + gold);

}

/// <summary>

/// Spends the gold if it is.

/// </summary>

/// <returns><c>true</c>, if gold was spent, <c>false</c> otherwise.</returns>

/// <param name="cost">Cost.</param>

public bool SpendGold(int cost)

{

bool res = false;

int currentGold = GetGold();

if (currentGold >= cost)

{

SetGold(currentGold - cost);

res = true;

}

return res;

}

/// <summary>

/// Sets the defeat attempts.

/// </summary>

/// <param name="attempts">Attempts.</param>

public void SetDefeatAttempts(int attempts)

{

defeatAttempts.text = attempts.ToString();

}

/// <summary>

/// On unit killed by other unit.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void UnitKilled(GameObject obj, string param)

{

// If this is enemy

if (obj.CompareTag("Enemy") || obj.CompareTag("FlyingEnemy"))

{

Price price = obj.GetComponent<Price>();

if (price != null)

{

// Add gold for enemy kill

AddGold(price.price);

}

}

}

/// <summary>

/// Buttons pressed handler.

/// </summary>

/// <param name="obj">Object.</param>

/// <param name="param">Parameter.</param>

private void ButtonPressed(GameObject obj, string param)

{

switch (param)

{

case "Pause":

GoToPauseMenu();

break;

case "Resume":

GoToLevel();

break;

case "Back":

ExitFromLevel();

break;

case "Restart":

RestartLevel();

break;

}

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

StopAllCoroutines();

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// Controls the sound effects and soundtrack volume via sliders.

/// </summary>

public class VolumeControl : MonoBehaviour

{

// Slider for sound effects volume

public Slider sound;

// Slider for soundtrack volume

public Slider music;

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

Debug.Assert(sound && music, "Wrong initial settings");

sound.value = DataManager.instance.configs.soundVolume;

music.value = DataManager.instance.configs.musicVolume;

sound.onValueChanged.AddListener(delegate {OnVolumeChanged();});

music.onValueChanged.AddListener(delegate {OnVolumeChanged();});

}

/// <summary>

/// Raises the volume changed event.

/// </summary>

private void OnVolumeChanged()

{

// Store new settings

DataManager.instance.configs.soundVolume = sound.value;

DataManager.instance.configs.musicVolume = music.value;

DataManager.instance.SaveGameConfigs();

// Apply new settings

AudioManager.instance.SetVolume(DataManager.instance.configs.soundVolume, DataManager.instance.configs.musicVolume);

}

}

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.UI;

/// <summary>

/// Timer to display current enemy wave.

/// </summary>

[RequireComponent(typeof(Image))]

public class WavesTimer : MonoBehaviour

{

// Visualisation of remaining TO

public Image timeBar;

// Current wave text field

public Text currentWaveText;

// Max wave text field

public Text maxWaveNumberText;

// Effect of highlighted timer

public GameObject highlightedFX;

// Duration for highlighted effect

public float highlightedTO = 0.2f;

// Waves descriptor for this game level

private WavesInfo wavesInfo;

// Waves list

private List<float> waves = new List<float>();

// Current wave

private int currentWave;

// TO before next wave

private float currentTimeout;

// Time counter

private float counter;

// Timer stopped

private bool finished;

/// <summary>

/// Raises the disable event.

/// </summary>

void OnDisable()

{

StopAllCoroutines ();

}

/// <summary>

/// Awake this instance.

/// </summary>

void Awake()

{

wavesInfo = FindObjectOfType<WavesInfo>();

Debug.Assert(timeBar && highlightedFX && wavesInfo && timeBar && currentWaveText && maxWaveNumberText, "Wrong initial settings");

}

/// <summary>

/// Start this instance.

/// </summary>

void Start()

{

highlightedFX.SetActive(false);

waves = wavesInfo.wavesTimeouts;

currentWave = 0;

counter = 0f;

finished = false;

GetCurrentWaveCounter();

maxWaveNumberText.text = waves.Count.ToString();

currentWaveText.text = "0";

}

/// <summary>

/// Update this instance.

/// </summary>

void FixedUpdate()

{

if (finished == false)

{

// Timeout expired

if (counter <= 0f)

{

// Send event about next wave start

EventManager.TriggerEvent("WaveStart", null, currentWave.ToString());

currentWave++;

currentWaveText.text = currentWave.ToString();

// Highlight the timer for short time

StartCoroutine("HighlightTimer");

// All waves are sended

if (GetCurrentWaveCounter() == false)

{

finished = true;

// Send event about timer stop

EventManager.TriggerEvent("TimerEnd", null, null);

return;

}

}

counter -= Time.fixedDeltaTime;

if (currentTimeout > 0f)

{

timeBar.fillAmount = counter / currentTimeout;

}

else

{

timeBar.fillAmount = 0f;

}

}

}

/// <summary>

/// Gets the current wave timeout.

/// </summary>

/// <returns><c>true</c>, if current wave timeout was gotten, <c>false</c> otherwise.</returns>

private bool GetCurrentWaveCounter()

{

bool res = false;

if (waves.Count > currentWave)

{

counter = currentTimeout = waves[currentWave];

res = true;

}

return res;

}

/// <summary>

/// Highlights the timer coroutine.

/// </summary>

/// <returns>The timer.</returns>

private IEnumerator HighlightTimer()

{

highlightedFX.SetActive(true);

yield return new WaitForSeconds(highlightedTO);

highlightedFX.SetActive(false);

}

/// <summary>

/// Raises the destroy event.

/// </summary>

void OnDestroy()

{

StopAllCoroutines();

}

}