***2034440***

Unity editor (FruitTilemap and PPTilemap don’t actually have a use right now, I literally just added them. They are copies of PelletTilemap which I was working from before adding the others the make the map look a lil nicer for u <3:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedTileData isn’t used, just kept there in case I do change my mind with implementation

A screenshot of a computer

Description automatically generatedscreenshots of A screenshot of a computer

Description automatically generatedpac-guy scranning (eating) (he can also only eat white\_pellets at the moment don’t judge him)

CODE:

MazeMover.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Serialization;

using UnityEngine.Tilemaps;

public class MazeMover : MonoBehaviour

{

    // Start is called before the first frame update

    void Start()

    {

        //Set our initial target position to be our starting position so

        //that the Update() will update the target position correctly.

        targetPos = transform.position;

        //This only works so long as there's only ONE TileMap

        wallTileMap = GameObject.FindObjectOfType<Tilemap>();

        //Quite heavy in terms of data so definitely don't want to call

        //on each Update() but its fine to do once in Start() for now.

    }

    float Speed = 3;

    private Vector2 desiredDirection; //current direction we want to move in

    private Vector2 targetPos;

    private Tilemap wallTileMap;

    // Update is called once per frame

    void Update()

    {

        //we have some kind of direction/velocity being applied (a force)

        //so move in that direction...

        //.. IF WE CAN. What if theres a wall in the way? Then we stop.

        //First check we can legally move in the direction we want!

        UpdateTargetPosition();

        //Do move.

        MoveToTargetPosition();

    }

    //INFO

    //Our objects are not physics-enabled rigidbodies, so

    //the physics system isn't  moving us, nor are we doing

    //'real' collisions, so Update() is more appropriate in this case.

    //INFO

    //Time.deltaTime is the time its been since the last call to what

    //we're currently (if we're in Update() its the time since this was

    //last called, if in FixedUpdate() its the time since that.

    void UpdateTargetPosition( bool force = false )

    {

        if (force == false)

        {

            //Have we reached our target?

            float distanceToTarget = Vector3.Distance(transform.position, targetPos);

            //Only checking for 0 here since we can check for this in MoveToTargetPosition()

            //and if we're a little bit out we can just set ourselves correctly there.

            if(distanceToTarget > 0)

            {

                //Not there yet, no need to update.

                return;

            }

        }

        //We have reached our target, we need a new target position.

        targetPos += desiredDirection;

        //Normalise the target position to a tile's position

        targetPos = FloorPosition(targetPos);

        if (isTileEmpty(targetPos))

        {

            return;

        }

        //if we get here it means our target position is occupied, so don't allow.

        targetPos = transform.position;

    }

    Vector2 FloorPosition(Vector2 pos)

    {

        //Normalising to a tile's position.

        //This might not line up right if we have a weirdly offset tilemap

        //A 'more robust' way to do this might be

        //to use the Tilemap's CellToWorld(), where you'd lookup the tile at the new

        //target position, reading back that Tile's world position.

        return new Vector2(Mathf.FloorToInt(pos.x), Mathf.FloorToInt(pos.y));

    }

    bool isTileEmpty(Vector2 pos)

    {

        return GetTileAt(pos) == null;

    }

    TileBase GetTileAt(Vector2 pos)

    {

        //First we need to change the world position to a tile cell index.

        Vector3Int cellPos = wallTileMap.WorldToCell(pos);

        //Now return the tile at that cell.

        return wallTileMap.GetTile(cellPos);

    }

    void MoveToTargetPosition()

    {

        //How far can we move this frame?

        float distanceThisUpdate = Speed \* Time.deltaTime;

        //And in what direction is this movement?

        //Towards our target position!

        //Also, we're giving the vector a length of 1 with 'normalized' \*spit\* silly americans \*in overtly british voice\*

        //to make it easier to work with when manipulating with arithmetic.

        Vector2 distToTarget = (targetPos - (Vector2)transform.position);

        //And how far are we moving in this update?

        Vector2 movementThisUpdate = distToTarget.normalized \* distanceThisUpdate;

        //What if we're moving PAST the target?

        //We COULD change movementThisUpdate to have the same magnitude as distance to target

        //We also don't care about the actual length of the vectors as we're comparing two, so

        //we can save some time in terms of maths here by comparing the squares of the magnitudes.

        if(distToTarget.SqrMagnitude() < movementThisUpdate.SqrMagnitude())

        {

            //We're past our target, so just move to it.

            transform.position = targetPos;

            return;

        }

        //Do Move!

        transform.Translate(movementThisUpdate);

    }

    public void SetDesiredDirection(Vector2 newDir)

    {

        //Just set our desired direction.

        //Make sure not diagonal? In THEORY, our PlayerMover/EnemyMover script already does this.

        // But we shouldn't accept a direction that would slam us into a wall.

        Vector2 testPos = FloorPosition(targetPos + newDir);

        if(isTileEmpty(testPos) == false)

        {

            //Trying to slam into wall, ignore.

            return;

        }

        Vector2 oldDir = desiredDirection;

        desiredDirection = newDir;

        //If the input is to reverse our direction, do it instantly?

        //if ((oldDir.x \* newDir.x) < 0 || (oldDir.y \* newDir.y) < 0)

        if (Vector2.Dot(oldDir, newDir) < 0) // above and this are mathematically identical hence going with slicker ver.

        {

            UpdateTargetPosition(true); //this is all we want to do, but

            //only when we're trying to reverse our direction.

        }

    }

    public Vector2 GetDesiredDirection()

    {

        //Get input!

        //float horizontal = Input.GetAxis("Horizontal");

        //float vertical = Input.GetAxis("Vertical");

        //desiredDirection = new Vector2(horizontal, vertical);

        return desiredDirection;

    }

}

PlayerMover.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Serialization;

//Not a Runtime 'thing', more so we can spot errors in the editor as they happen easier,

//also to ensure the GameObject we want to have this component DEFINITELY DOES have it.

[RequireComponent(typeof(MazeMover))]

public class PlayerMover : MonoBehaviour

{

//Could subclass and inherit Update from MazeMover but having this as just its own

//implementation is how I'm going to do it as I'm following along a tutorial and

//its how theyve explained it, but please let me know if you have any thoughts.

    // Start is called before the first frame update

    void Start()

    {

        mazeMover = GetComponent<MazeMover>();

    }

    MazeMover mazeMover;

    // Update is called once per frame

    void Update()

    {

        //Even if we're performing our movement in th FixedUpdate we still need to

        //capture the current position here to update the target position correctly.

        //Usually we want to do this as its a good force of habit to prevent falling

        //into mistakes using GetKeyUp or GetKeyDown in FixedUpdate, is my understanding of it.

        /\* if (Input.GetKey(KeyCode.LeftArrow))

        {

            //Returns true EVERY FRAME while the key is held down

        }

        if (Input.GetKeyDown(KeyCode.LeftArrow))

        {

            //Returns true on the first VISUAL frame when the key is first pressed down

        }

        if (Input.GetButton('Left'))

        {

            //could define this 'Left' in the Input Manager in Unity

        }\*/

        //In this situation, to follow along with tutorial we'll use GetAxisRaw since we want

        //it to properly tell us the exact value of the button press (straight to 1 instead of

        //simulating a joystick's axis movement with GetAxis)

        Vector2 newDir = new Vector2(

            Input.GetAxisRaw("Horizontal"),

            Input.GetAxisRaw("Vertical")

        );

        if (newDir.SqrMagnitude() < 0.05f)

        {

            //This is effectively 0, so just return and do nothing since no input is happening

            //If you have any thoughts on a more robust solution such as using Mathf.Abs, please let me know.

            return;

        }

        // newDir could be REALLY wonky at this point. Could be diagonal, could have

        //a fractional number like (0.67, -0.24) just for an example.

        //So, we want to sanitise this value to drop 1 of the axes if there is 2 of them:

        //In case we have both an X and Y.

        if (Mathf.Abs(newDir.x) > Mathf.Abs(newDir.y))

        {

            newDir.y = 0;

        }

        else

        {

            newDir.x = 0;

        }

        mazeMover.SetDesiredDirection(newDir.normalized);

        //Or:

        /\* if (Input.GetAxisRaw("Horizontal") < 0)

        {

            //Returns a value from -1 to 1 based on left/right input

            mazeMover.SetDesiredDirection(newDir.left);

        }

        if (Input.GetAxisRaw("Horizontal") > 0)

        {

            mazeMover.SetDesiredDirection(Vector2.right);

        }

        if (Input.GetAxisRaw("Vertical") < 0)

        {

            mazeMover.SetDesiredDirection(Vector2.down);

        }

        if (Input.GetAxisRaw("Vertical") > 0)

        {

            mazeMover.SetDesiredDirection(Vector2.up);

        } \*/

    }

}

PelletTilemap.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Tilemaps;

public class PelletTileMap : MonoBehaviour

{

    // Start is called before the first frame update

    void Start()

    {

        pelletEaters = GameObject.FindObjectsOfType<PelletEater>();

        myTilemap = GetComponent<Tilemap>();

        //easy, because the script belongs to the same object as the Tilemap

    }

    // TODO: Add code for pellet eaters to signal updating this whenever they come or go.,

    //if that's something that happens (such as adding or removing PelletEaters (multiplayer?))

    PelletEater[] pelletEaters;

    Tilemap myTilemap;

    //'FruitTilemap' and 'PPTilemap' created in Unity. (Identical to 'PelletTilemap' (used to hold

    //reference to this script, and to hold the sprites for fruits and power pellets (PP))).

    //What happens when we eat a pellet on this map?

    //For a simple Pac-Man game, this is all we'd really need to know, but could be expanded on and

    //potentially re-done into a more flexible but more complex design.

    public int PelletPoints = 1;

    public bool RequiredForLevelCompletion = false;

    public float PowerSeconds = 0;

    // Update is called once per frame

    void Update()

    {

        //Is a pellet eater on a tile with a pellet?

        foreach (PelletEater pe in pelletEaters)

        {

            CheckPellet(pe);

        }

    }

    void CheckPellet(PelletEater pelletEater)

    {

        Vector2 offsetPos = (Vector2)pelletEater.transform.position + new Vector2(0.5f, 0.5f);

        //TODO: Add code to check what tile pe is in, and if there is a pellet there.

        TileBase tile = GetTileAt(offsetPos);

        if (tile == null)

        {

            //Empty tile with no pellets.

            return;

        }

        Debug.Log("Scran!");

        EatPelletAt(offsetPos);

    }

    void EatPelletAt(Vector2 pos )

    {

        //TODO: Add code to eat the PelletEater's Pellet

        SetTileAt(pos, null);

    }

    void SetTileAt(Vector2 pos, TileBase tile)

    {

        Vector3Int cellPos = myTilemap.WorldToCell(pos);

        myTilemap.SetTile(cellPos, tile);

    }

    //below is ctrl c+v from MazeMover.cs

    TileBase GetTileAt(Vector2 pos)

    {

        //First we need to change the world position to a tile cell index.

        Vector3Int cellPos = myTilemap.WorldToCell(pos);

        //Now return the tile at that cell.

        return myTilemap.GetTile(cellPos);

    }

}

//Before PelletTileMap.cs existed there was Pellet.cs, and the consideration was to just

//code the behaviour of when a Player touches a Pellet there. But, this would be too

//challenging to handle accurately ourselves, so then the consideration was to use Unity's

//2D collision/trigger component within the Pellet -> Graphics so that complex partitioning

//isn't an issue I have to deal with directly. But, from my understanding, this would still

//require a good bit of computing, so the consideration now is to have all the logic within

//TileMap itself, instead.

//From old Pellet.cs

    //Could use the collision system Unity gives us to do this

        /\* void OnTriggerEnter2D(Collider2D other)

        {

            //and then something like:

            if (other.gameObject.tag == "Pellet")

            {

                Destroy(other.gameObject);

            }

        } \*/

        //But using this system might be overkill for this scope, so we'll just do a

        //more code-centric way than above.

        //But if we want the game to have all the pellets and all the pelletEaters always

        //checking for one another's distance to each other, we'd need some pretty complex

        //partitioning code to be written, which is definitely outside the scope of this project.

        //pelletEaters = GameObject.FindObjectsOfType<PelletEater>();

        //and then having a for loop then goes through a global array of all pelletEaters for example,

        //would be pretty slow presumably.

        //So, we'll go back to Unity's collision system.

        //At the point of this comment, we want the pellet to be part of the tilemap system,

        //so its easier to paint out for 1, and also can't have more than 1 pellet per tile.

PelletEater.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

public class PelletEater : MonoBehaviour

{

    // Start is called before the first frame update

    void Start()

    {

    }

    // Update is called once per frame

    void Update()

    {

    }

}

TilePellet.cs

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Tilemaps;

//Inherits from TileBase rather than MonoBehaviour since we're trying to subclass from

//TileBase.

public class TilePellet : TileBase

{

}

TileData.cs (not used but has my horribly laid out thoughts inside)

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

using UnityEngine.Tilemaps;

[CreateAssetMenu(fileName = "TileData", menuName = "ScriptableObjects/TileData", order = 1)]

public class TileData : ScriptableObject

{

    //The consideration here is that we can create instances of this scriptable object that

    //only held the graphics information (the sprite) which a manager class could use to check

    //whether a player has 1. collided with a wall or 2. collided with a pellet.

    //In the future this could be useful to trigger some kind of 'event' when a player collides

    //with a pellet specifically (not a wall since the only action taken is preventing the player

    //from entering that tile). This could then be expanded on to implement extra mechanics

    //for the player interacting with 'things' that give the player extra points or extra lives for

    //example.

    //I believe that would be the 'observer pattern'.

    //I'm not sure if this is the best way to do it, but the way I'm going to go for now, with the

    //PelletTileMap and TilePellet is similar to this, without the ScriptableObjects. In fact, the

    //pellets aren't even game objects, they're just sprites on a tilemap. So, the Tilemaps will

    //be the 'observer' and the sprites attached to them will be the things that are being observed.

    // TODO: I think maybe subclassing from TileData would be the way to go if I wanted to add my own mechanics in the

    //future (like powerups that arent in Pac-Man for example).

    //I want to use this class only to hold the information of which sprite is displayed

    //within it.

    public Sprite sprite;

}