Game Programming

Lecture 3: Level and Scene Loading

December 19, 2022

Paul Bonsma

Last Week - Poll

- Game objects, hierarchy, collision checking, type checking and casting
- Creating game play: (collision based) interaction between objects

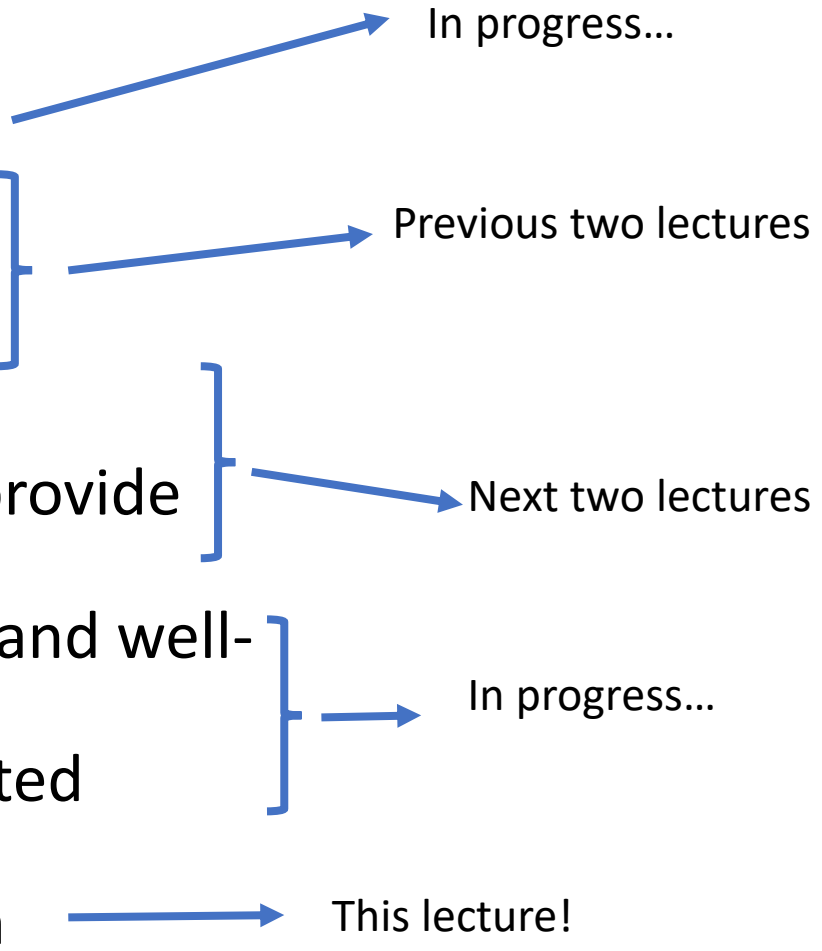
- Do you already have a controllable player character and interactions with other objects? (Enemies, pickups, walls, bullets, …)
  1. Collisions? Game objects?
  2. Just getting started…
  3. Almost
  4. Yes!
  5. I’m already adding more than that to my game

This Week

- Next steps:
  - Creating tooling for level loading and creation
  - Switching levels / scenes
  - Managing complexity / keeping the project clean

Course Learning Goals

1. …create a fully functional game, including
   a) creating, removing and manipulating objects
   b) keyboard and mouse input
   c) collision handling between two objects
2. …apply visual and auditory effects
3. …design and implement a simple HUD to provide player feedback
4. …apply code conventions for writing clear and well-structured code
5. …apply the basic concepts of Object Oriented Programming (OOP)
6. …create tooling to enable content creation



Lecture Outline

- Managing Complexity (Access Modifiers, Encapsulation, Namespaces)
- Basic Level Loading (Arrays, Lists)
- Advanced Level Loading (Tiled)
- Keeping track of Data (Static, alternatives)
- Conclusions

Managing Complexity

Folders, namespaces, encapsulation, access modifiers, properties

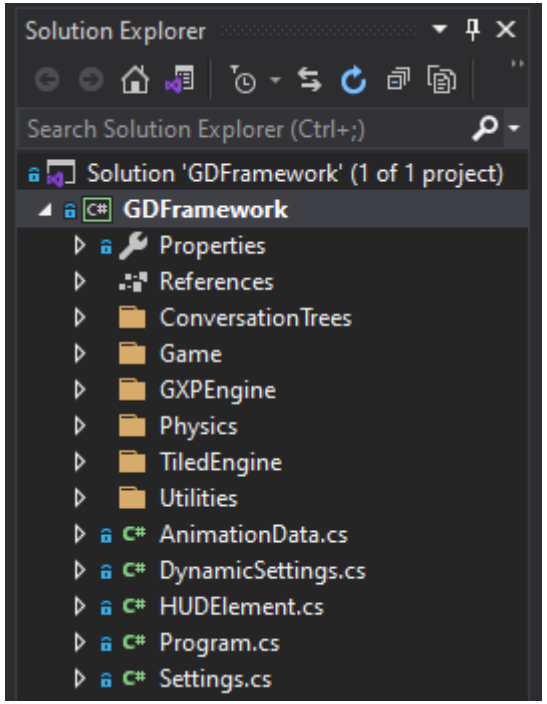
Managing Complexity

- As mentioned in the first lecture, the biggest challenge in programming is not learning the engine commands, functionality, or language syntax, but *managing complexity* as your project grows

- Currently your project is growing – it's time to talk about some techniques

Folders

- As your project grows, it's good to divide your scripts into folders – groups of scripts that belong together
- In Visual Studio:
  - In solution explorer: right-click your project → Add → New Folder
  - Drag your scripts into the proper folder



Namespaces - How

- You can put your classes into *namespaces*
- To access (use) a class *MyClass* in a namespace *MySpace*:
  - Make sure your code is in the same namespace
  - Add `using MySpace;` on top of the file
  - Preface the class with the namespace: `MySpace.MyClass`

```
using System;
using GXPEngine.Core;

namespace GXPEngine
{
    /// <summary>
    /// The Sprite class holds 2D images
    /// </summary>
    34 references
    public class Sprite : GameObject
    {
```

- Examples:
  - The class *Console* lives in the *System* namespace
  - Most *GXPEngine* classes (Sprite, GameObject, …) live in the *GXPEngine* namespace

Namespaces - Why

- Namespaces prevent *name clashes* in a growing project
  - For instance, both the GXPEngine and your game can contain a *Camera* class
  - Distinguish between them by typing GXPEngine.Camera / MyGame.Camera

- More subtle: it helps you prevent accidental *coupling* between classes: you need an explicit *using* statement to use a class from another namespace
  - Example: you don't want to create a *dependency* between your general-purpose tooling classes and your game-specific code!

- It's good practice to have your namespaces correspond to your folders

Access Modifiers: Public and Private

- You can add *public* or *private* in front of methods and fields (variables)
- These are *access modifiers:* only public things can be accessed from another class
- The default is private, so that may be omitted(!)

- There are more access modifiers (later we'll see *protected*), applications, and exceptions to these rules (outside the scope of this course)

Public/private example

Private by default

From outside this class, we can only call *SetTarget* (and the constructors)

That's exactly what we want!

```
int lastShootTime = 0;

private Sprite target;

1 reference
public Shooter() : base("Plant.png", 6, 4) {
    Initialize();
}
0 references
public Shooter(string imageFile, int cols, int rows) : base(imageFile, cols, rows) {
    Initialize();
}

2 references
void Initialize() {
    SetOrigin(width / 2, height / 2);
    SetCycle(12, 7); // idle cycle
    //SetCycle(0, 12); // shoot cycle
    collider.isTrigger = true;
}

1 reference
public void SetTarget(Sprite target) {
    this.target = target;
}
```

Interface vs Implementation

- All your public methods and fields form the *interface* of the class
  - Interfaces should rarely change
  - Interfaces should be kept small (in particular: you should rarely make *variables* public)
  - It's your responsibility to make sure the class still works *correctly*, when interface methods are called in unexpected ways!
- Private methods and fields form the *implementation*
  - May change (e.g. make more efficient, add logging functionality, improve code quality), without changing the interface

In practice

- While writing a class, think about what can happen if you're working together with another (possibly mediocre) programmer on a project: *what are all the things that could go wrong?*

- A well written class doesn't need a `manual', to be *used* by others! (=robust)

- It may need some explanation if it has to be *modified* by others, but that shouldn't be necessary too often

Encapsulation

- *Encapsulation* is one of the cornerstones of OOP:
  - Bundle *data* (=variables) and *behaviors* (=methods) of an *object* together in a class
  - The class is responsible for maintaining the correctness of its *state* (=variable values)
  - Separate *interface* from *implementation*

- In short:
  - *Think twice before you make a method public!*
  - (Think thrice before you make a *variable* public!!)

Properties

- C# allows *properties* (or getters / setters): from outside, these look like public variables, but they are more like methods.

- You can use these to do extra
  - Error checking
  - Administration / logging

- You can also *omit* the public setter

```
public class PlayerData {
    const int startLives = 3;

    // default access modifier = private:
    int _lives = 0;

    3 references
    public int lives {
        get {
            return _lives;
        }
        set {
            int oldLives = _lives;
            _lives = value;
            if (_lives<0) {
                _lives = 0;
                Console.WriteLine(
                    "Warning: lives cannot be negative: was {0}, new value: {1}.",
                    oldLives,value);
            }
            Console.WriteLine("Player lives: " + _lives);
        }
    }

    1 reference
    public PlayerData() {
        Reset();
    }

    2 references
    public void Reset() {
        _lives = startLives;
        Console.WriteLine("Resetting player data. Lives = {0}",_lives);
    }
}
```

String Formatting

- C# allows *string formatting* in *Console.WriteLine* or *String.Format*:
- Variable values are inserted at the positions {0}, {1}, etc.
- There are many more options… (Look them up if needed)

```
Console.WriteLine(
    "Warning: lives cannot be negative: was {0}, new value: {1}.",
    oldLives,value);
```

Basic Level Loading

Project Introduction, Arrays, Lists, Level Class

Updated Project

- We start with last week's lecture demo. Changes:
  - Shooters have a *target* (the player): they only shoot when the player is in front of them
    - After the Shooter has been constructed, its public *SetTarget* method needs to be called
    - Note: the class still works if no target is set! (Robustness)
  - The player has *platformer controls:*
    - Gravity
    - Jumping
    - No double jumping: can only jump when *grounded*

- Checking whether the player is grounded requires inspecting the *Collision* object returning by MoveUntilCollision (namespace: *GXPEngine.Core*)

Platformer Controls

```
float dx = GetHorizontalInput();

vy += gravity;
if (grounded && Input.GetKeyDown(Key.SPACE)) { // no double jump
    vy = -jumpStrength;
}
grounded = false;

// Note: other objects need to have trigger colliders:
MoveUntilCollision(dx, 0);

// using GXPEngine.Core   for Collision:
Collision col = MoveUntilCollision(0, vy);
if (col!=null) {
    vy = 0;
    if (col.normal.y < 0) { // only grounded when on floor!
        grounded = true;
    }
}

CheckCollisions();
```

Last Week

```
public MyGame() : base(320, 256, false, false, 960 , 768, true)
{
    level = new Pivot();
    AddChild(level);


    CreateBackground();
    CreateWalls();
    CreateUI();

    player = new Player();
    player.SetXY(128, 192);
    level.AddChild(player);

    Enemy enemy = new Enemy();
    enemy.SetXY(64, 128);
    level.AddChild(enemy);

    Pickup pickup = new Pickup();
    pickup.SetXY(64, 64);
    level.AddChild(pickup);

    Shooter plant = new Shooter();
    plant.SetXY(260, 5*16 - plant.height/2);
    level.AddChild(plant);

    // Add UI last (on top):
    AddChild(healthUI);
    AddChild(ammoUI);
}
```

Tooling Goal

- Doing it like last week is *insufficient*! (Grading criteria)

- Your main goal is to *enable artists, level and game designers* (even though right now you're both the programmer and designer)
- You need at least *2D arrays*
- Ideally, you want good tooling:
  - Quick iteration for level creation
  - Quick parameter changes, game play tweaking (e.g. enemy speed)
  - Quick iteration for art (animations, sounds, tiles)
  - Flexibility (e.g. quick change in level order)
  - *Hot reloading:* change the level while the program is still running!

Arrays

- Arrays have fixed length (in C#: *Length* property)
- We have seen them before (GetCollisions)
- Basic type: light weight, but inflexible
- Initialization: see below

```
// collision checking:
GameObject[] collisions = GetCollisions();
for (int i=0;i<collisions.Length;i++) {
    if (collisions[i] is Solid) {
        TurnAround();
        x = oldX;
        break;
    }
}
```

```
int[] myArray1 = new int[5];
int[] myArray2 = new int[5] { 1, 2, 3, 4, 5 };
int[] myArray3 = { 1, 2, 3, 4, 5 };
```

Multi dimensional Arrays

- Multi-dimensional arrays are declared with commas between the brackets
- Use GetLength(0), GetLength(1), etc. to get the dimensions
- These can be used to create your levels in a slightly better way

```
int[,,] levels = // new int[2,7,9] // this is optional,
{
    // level 1:
    {
        {1,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,3,1},
        {1,0,0,0,0,0,0,1,1},
        {1,0,2,0,0,0,0,0,1},
        {1,1,1,1,1,1,1,1,1}
    },
    // level 2:
    {
        {1,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,0,1},
        {1,2,0,0,0,0,0,0,1},
        {1,1,0,0,0,0,0,0,1},
        {1,0,0,0,0,0,0,3,1},
        {1,1,1,1,1,1,1,1,1}
    }
};
```

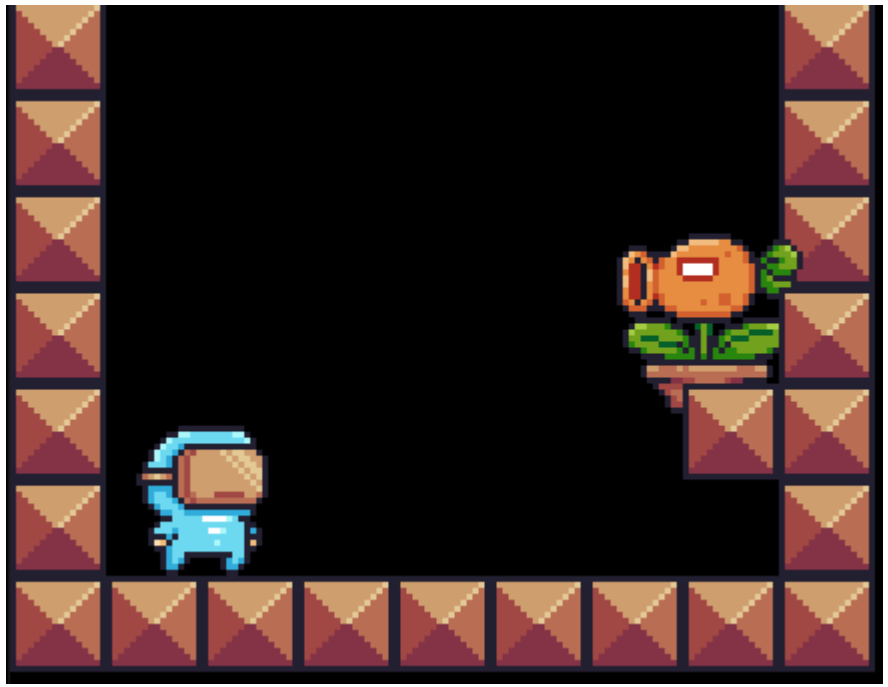
Level Initialization using Arrays

```
int tileSize = 16;
Player player=null;
for (int row=0;row<levels.GetLength(1);row++) {
    for (int col=0;col<levels.GetLength(2);col++) {
        int tileType = levels[index, row, col];
        switch (tileType) {
            case TILE:
                Solid tile = new Solid();
                tile.x = col * tileSize;
                tile.y = row * tileSize;
                AddChild(tile);
                break;
            case PLAYER:
                player = new Player();
                // TODO: fine tune the placement!
                player.x = col * tileSize;
                player.y = row * tileSize;
                AddChild(player);
                break;
            case SHOOTER:
                Shooter shooter = new Shooter();
                // TODO: fine tune the placement!
                shooter.x = col * tileSize;
                shooter.y = row * tileSize;
                AddChild(shooter);

                if (player!=null) {
                    Console.WriteLine("Setting shooter target");
                    shooter.SetTarget(player); // What if the shooter is created before the player?
                } else {
                    Console.WriteLine("Warning: creating shooter without target!");
                }
                break;
        }
    }
}
```

```
const int TILE = 1;
const int PLAYER = 2;
const int SHOOTER = 3;
```

```
// level 1:
{
    {1,0,0,0,0,0,0,0,1},
    {1,0,0,0,0,0,0,0,1},
    {1,0,0,0,0,0,0,0,1},
    {1,0,0,0,0,0,0,3,1},
    {1,0,0,0,0,0,0,1,1},
    {1,0,2,0,0,0,0,0,1},
    {1,1,1,1,1,1,1,1,1}
},
```



Level Switching - Setup

- As the MyGame class grows, we split it up into two classes with their own responsibilities:

- The *Level* class (a game object) is responsible for:
  - loading one particular level (from a file), creating game objects
  - Scrolling
- The *MyGame* class is responsible for *switching* levels:
  - Cleaning up (destroying) the old Level
  - Loading a new one

Level Switching in MyGame

```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.Destroy();
    }
}

6 references
public void LoadLevel(string filename) {
    DestroyAll();
    AddChild(new Level(filename));
    CreateUI();
}

0 references
void Update() {
    // Hot Reload:
    if (Input.GetKeyDown(Key.Q) && Input.GetKey(Key.LEFT_SHIFT)) {
        Console.WriteLine("Reloading + starting " + startLevel);
        LoadLevel(startLevel);
    }
```

Lists in C#

```
void DestroyAll() {
    List<GameObject> children = GetChildren();
    foreach (GameObject child in children) {
        child.Destroy();
    }
}
```

- GetChildren returns a List of GameObjects
- *Lists* are arrays that can *grow* and *shrink* (Java: ArrayList)
- Namespace: *System.Collections.Generic*
- Many useful methods/properties: *Count, Add, Clear, Remove, RemoveAt, IndexOf, Sort, ToArray* (and more… Visual Studio will show you all you need to know!)

- The weird brackets are *generics*: this is how you can *pass different types* to methods / classes, etc. (A list of *GameObjects*)
- *foreach* loops give an alternative for for-loops, for arrays and lists

Tiled

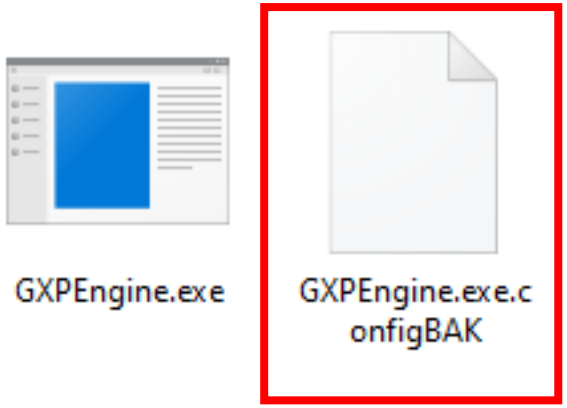
Editing levels / maps

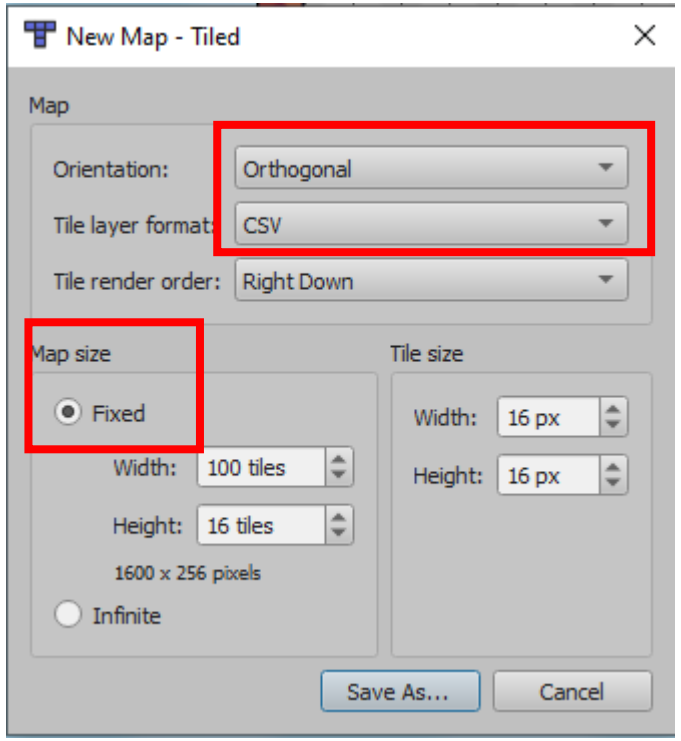
Tiled

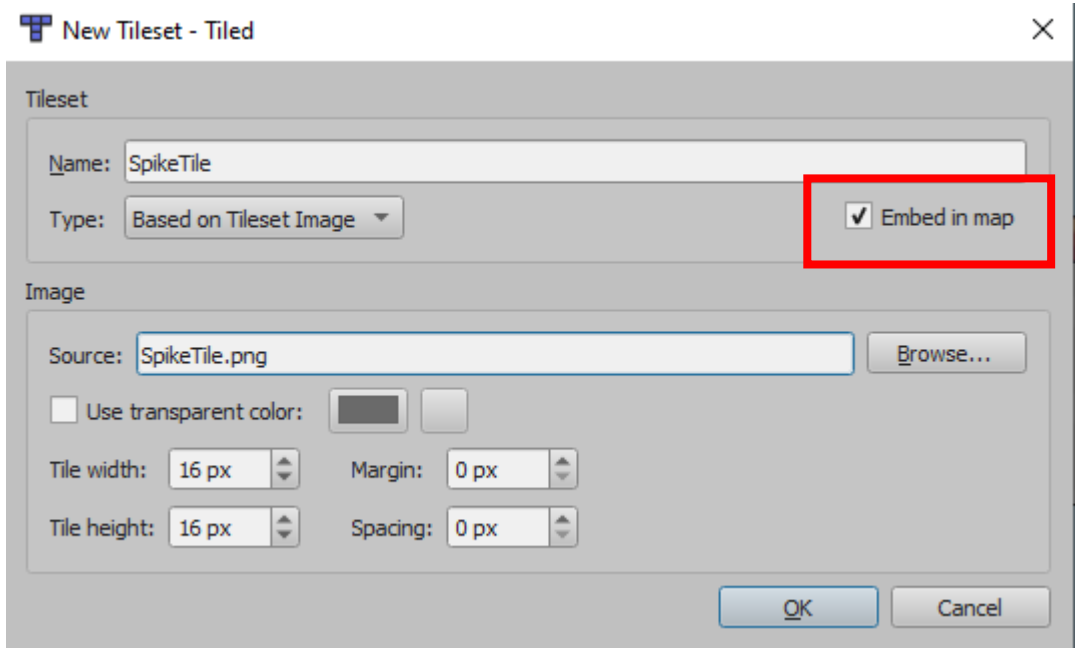
- *Tiled* is the level / scene / *map* editor for the GXPEngine
- You can get it from https://www.mapeditor.org/
- It supports
  - Tile layers
  - Object groups (=game objects)
  - Image layers
- It saves the map to an xml file
  - You can open it in a text editor too
  - Save it in bin/Debug, next to your assets (sprites)
- *Demo (GDF, workflow, basics)*

Tiled – Preventing Errors

- New map: *csv, orthogonal, fixed*
- New tile set: *embed in map*
- Tile sets: *must be in bin/Debug*
- Windows: delete or rename *GXPEngine.exe.config* !







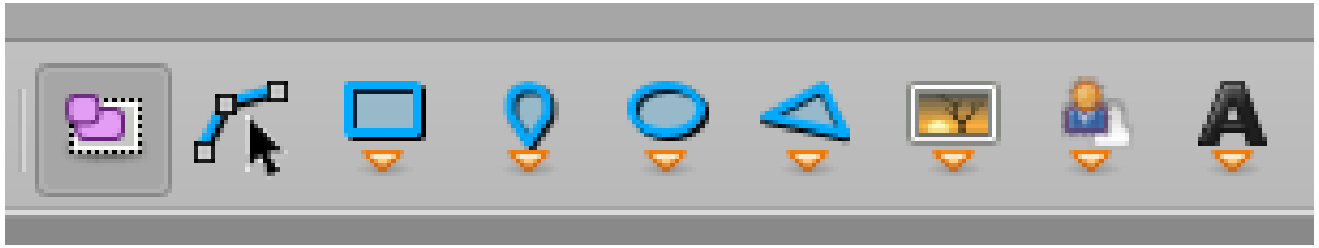
TiledLoader

- The easiest way to load Tiled files is using the *TiledLoader* class
  - Folder: *GXPEngine/AddOns*
  - Namespace: *TiledMapParser*

- For a custom approach, you can also use the *MapParser* class

- *Demo*: loading image layers / tile layers / object groups

Object Types

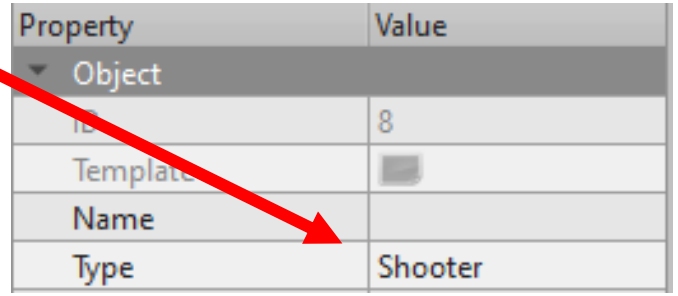


- In Tiled you can create three types of objects:
  - *Shape* objects → ?
  - *Image* objects → Will be *AnimationSprite*
  - *Text* objects → Will be *EasyDraw* (details: next week)

- If you want to create specific objects (Player, Pickup, etc.), you can set *autoInstance = true,* and specify the type in Tiled
- Only for shape objects and image objects

Automatic Object Creation: autoInstance

- In Tiled, the *Type* needs to be ClassName, or NameSpace.ClassName if namespace used
- Requires special *constructors* (note that you can add multiple constructors!):




```
// This constructor supports TiledLoader with autoInstance=true, for shape objects:
1 reference
public Shooter(TiledObject obj=null) : base("Plant.png", 6, 4) {
    Initialize(obj);
}
// This constructor supports TiledLoader with autoInstance=true, for image objects:
0 references
public Shooter(string imageFile, int cols, int rows, TiledObject obj = null) : base(imageFile, cols, rows) {
    Initialize(obj);
}
```

Auto Object Creation: Pitfalls & Challenges

- *Pitfall:* The TiledLoader will set the *animation frame, position, rotation, scale, origin* after the constructor of the object is done!
- *Challenge*: What about linking objects?
  - Example: the target of a shooter
  - The Shooter constructor has no reference to the current player or even level
  - Player may even be created after the shooter!
  - Solution: *Link the objects after creation*, using e.g. *FindObjectsOfType*

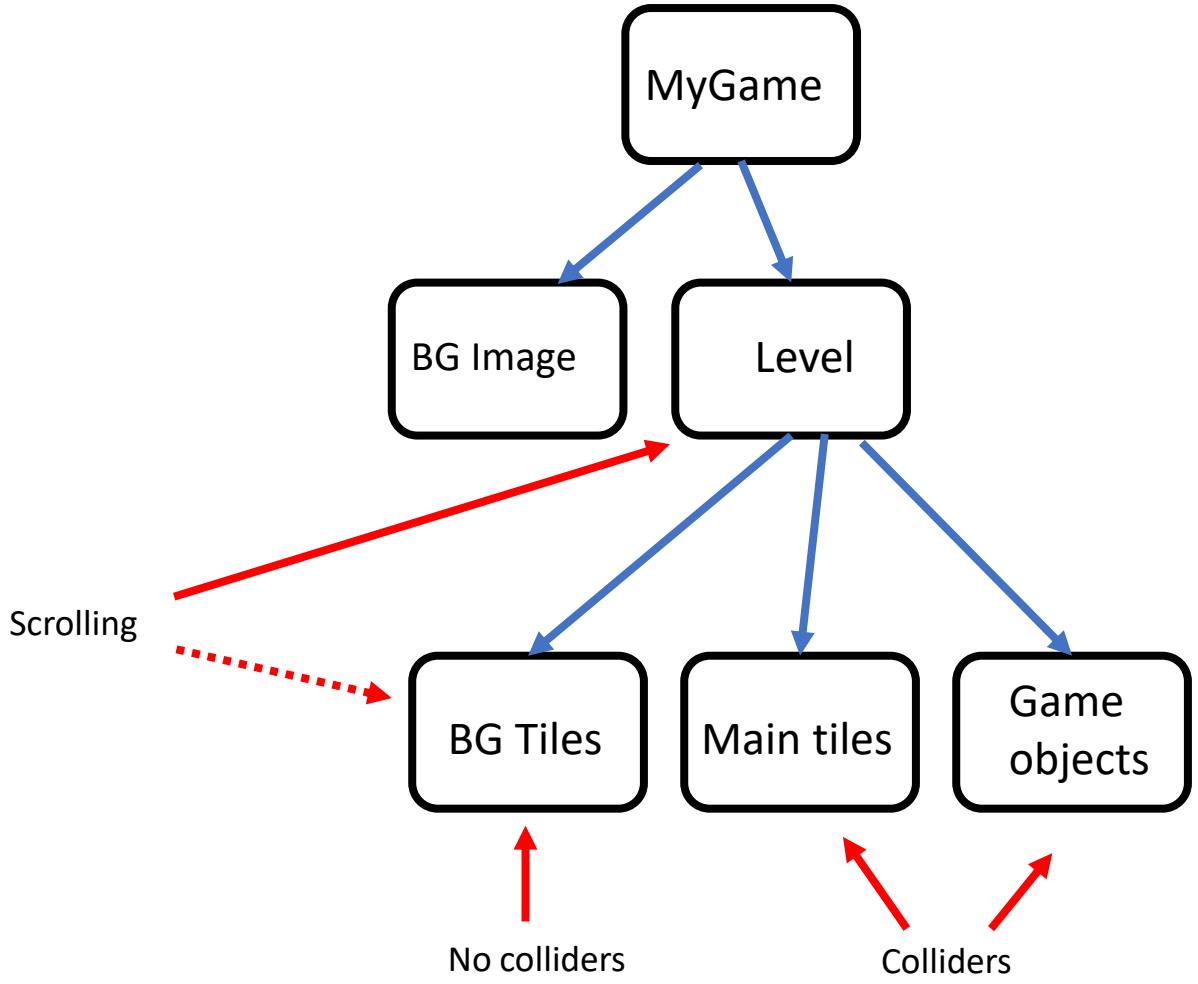
Finding Objects

- Using *someGameObject.FindObjectsOfType<SomeType>()* you can find all objects of a certain type that are descendant (child) of a given game object.
- (This is an example of generics again)
- It's relatively slow - don't overuse this! *(Don't use it in update!)*

```
    player = FindObjectOfType<Player>(); // Note: will fail if there are two players!
    if (player != null) {
        Shooter[] shooters = FindObjectsOfType<Shooter>();
        foreach (Shooter shooter in shooters) {
            shooter.SetTarget(player);
        }
    }
}
```

TiledLoader Properties

- Use *rootObject* to specify the root game object before loading (e.g. for scrolling)
- Use *addColliders* to specify whether the *tiles* should have colliders (e.g. background / foreground)

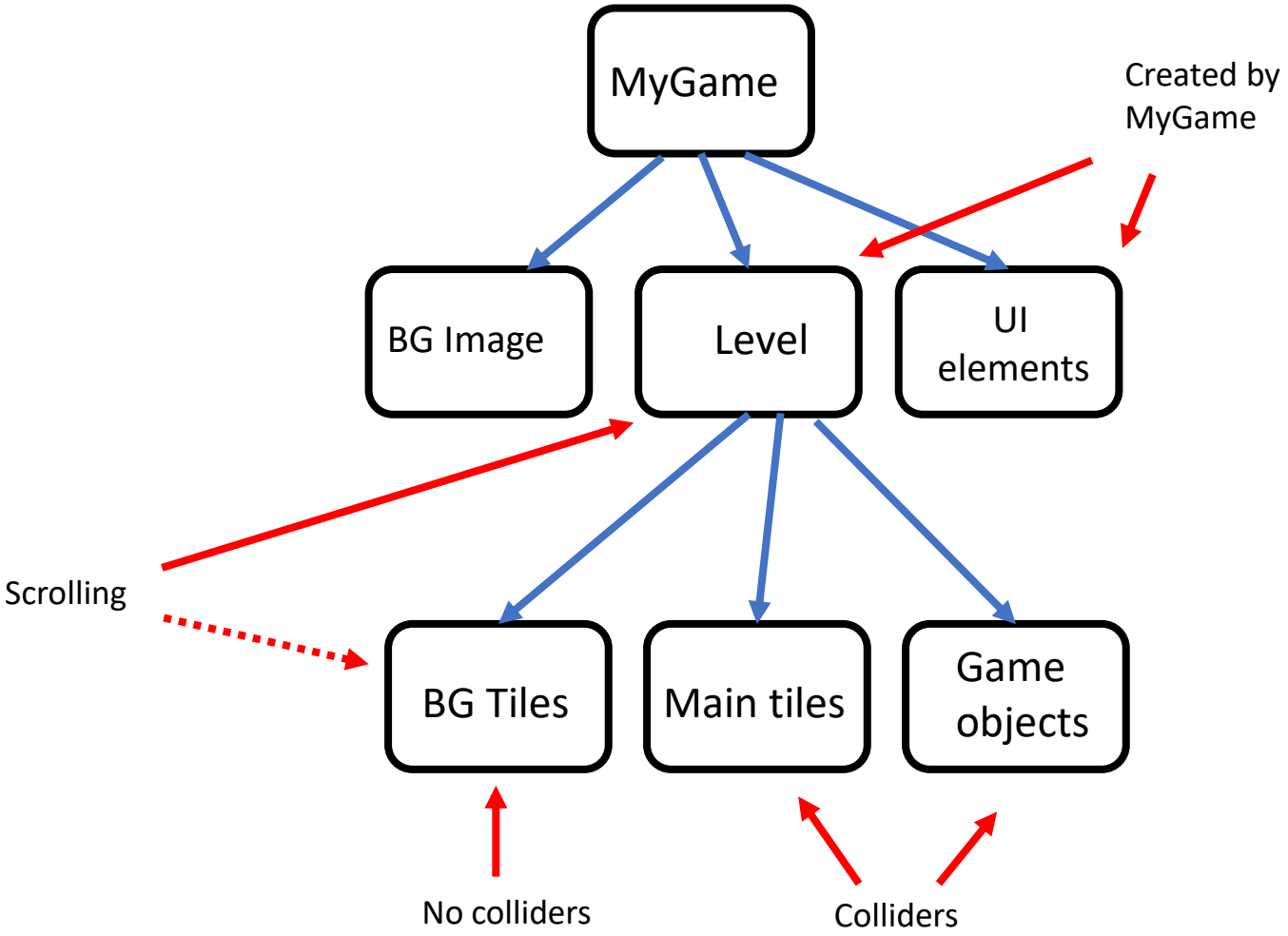


TiledLoader: Summary

```
void CreateLevel(bool includeImageLayers=true) {
    Console.WriteLine("Spawning level elements");
    loader.autoInstance = true; // automatically create Player, Pickup, Shooter, etc.
    loader.addColliders = false;
    loader.rootObject = game; // image is child of game - not scrolling
    if (includeImageLayers) loader.LoadImageLayers();
    loader.rootObject = this; // the rest is child of level - scrolling
    loader.LoadTileLayers(0); // background - no colliders
    loader.addColliders = true;
    loader.LoadTileLayers(1); // main layer - with colliders
    loader.LoadObjectGroups();

    // linking objects:
    player = FindObjectOfType<Player>(); // Note: will fail if there are two players!
    if (player != null) {
        Shooter[] shooters = FindObjectsOfType<Shooter>();
        foreach (Shooter shooter in shooters) {
            shooter.SetTarget(player);
        }
    }
}
```

Hierarchy Overview



Object Properties

```
// Configure from Tiled:
if (obj != null) {
    shootIntervalMs = obj.GetIntProperty("cooldownMs", 2000);
    bulletSpeed = obj.GetFloatProperty("bulletSpeed", 2);
}
```

- In Tiled, you can set *properties* for most things
  - Property containers: objects, layers, the map itself
  - Types: int, float, string, bool, color
- If you have a reference to the TiledObject, you can read these values
- *This is powerful* - your game designer will love it! Examples:
  - Quickly iterating on game play values (bullet speed, cooldown, …)
  - Specifying which level should be loaded next
- *Demo*

```
public Gate(string filename, int cols, int rows, TiledObject obj) : base(filename,cols,rows) {
    nextLevel = obj.GetStringProperty("nextLevel", "map1");
```

Tiled as User Interface Design Tool

- Tiled can also be used to design user interfaces, such as:
  - Main menu / options / high score / end / shop screen
  - In-game UI (HUD)
- TiledLoader loads text elements automatically (using system fonts)
- Useful: Making a Button class
- (demo)

```
class Button : GameObject {
    Sprite visualButton;
    string filename;

    1 reference
    public Button(Sprite visualButton, TiledObject obj) {
        this.visualButton = visualButton;
        filename = obj.GetStringProperty("load", "map1");
    }

    0 references
    void Update() {
        if (visualButton.HitTestPoint(Input.mouseX,Input.mouseY)) {
            visualButton.SetColor(1, 1, 1);
            if (Input.GetMouseButtonDown(0)) {
                ((MyGame)game).LoadLevel(filename + ".tmx");
            }
        } else {
            visualButton.SetColor(0.7f, 0.7f, 0.7f);
        }
    }
}
```

Keeping track of Data

Maintaining Data

- Levels are *destroyed* and *reloaded*, just like all their child game objects (like the player)
- Depending on the game design:
  - Some values should be reset every time (player health, ammo): *temporary*
  - Some values should be kept (player score, player lives): *persistent*
- Temporary values can be stored in the Level or Player class
- What about persistent values?
  - In MyGame? (Single Responsibility…)
  - static variables? (Hm…)
  - Something else?

Static

- The *static* keyword specifies that methods or fields are defined at *class level*, instead of at instance level.
  - Only *one copy* of such a variable per class
  - Calling / usage: ClassName.StaticMethod() or ClassName.StaticVariable
  - Example: *Console.WriteLine, Utils.Random, Game.main*
- Using *public static* variables, we can easily create variables that are persistent, and accessible from anywhere. (=*global* variables)
- …but should we…?

```
class Player : AnimationSprite {

    public static int lives; // Is this a good idea...?
```

Public static - Disadvantages

- It violates *encapsulation* (everyone can meddle with it!)
- Who is *responsible* for maintaining the values? Resetting them?
- *Extensibility:* what if you later want to have *two* versions of this data? (Two player game?)

Long story short:

- …it's easy at first, but a common source of bugs later!
- …if you do it too much, you won't pass this course!!

A Better Solution

- We use an instance of the *PlayerData* class (see the `Properties' slide)
- It contains some error handling code
- MyGame owns this copy (=responsible), and resets it when needed

- This class can be extended later (score, powerups, status effects, weapons, inventory, …)

Implementing Lives

- Level contains a PlayerDeath() method
- It’s called when the player dies (by the player, or by the level)
- Depending on the number of lives left, it
  - Reloads the current level, or
  - Loads the game over screen

Player Death / Respawning

```
public void PlayerDeath() {
    Console.WriteLine("Player dies");
    PlayerData data = ((MyGame)game).playerData;

    // Prevent die-twice-in-one-frame bug!:
    if (respawn || data.lives<=0) {
        Console.WriteLine("...for the second time in one frame - ignored");
        return;
    }

    data.lives--;

    if (data.lives <= 0) {
        ((MyGame)game).LoadLevel("menu.tmx");
    } else {
        respawn = true;
        ((MyGame)game).LoadLevel(currentLevelName);
    }
}
```

Conclusions

Public Interfaces

- MyGame:
  - playerData *(readonly)*
  - LoadLevel(string filename)
  - ShowHealth(int health)
  - ShowAmmo(int ammo)
- Level:
  - PlayerDeath()
- Shooter:
  - SetTarget(Sprite target)
- PlayerData:
  - lives *(a property)*
  - Reset()

Lifetimes, Responsibilities

- *MyGame*: only one throughout (as always)
- *Level*: destroyed / created by MyGame. Should be only one at the same time
- *PlayerData*: only one (for now), owned by MyGame
- *Specific game objects* (Shooter, Player, Pickup, …): created by Level, children of Level, destroyed when level is destroyed
- *UI elements:* children of MyGame, created and destroyed by MyGame

Summary

- Folders, namespaces
- Access modifiers (public, private), encapsulation
- Arrays and Lists
- Creating tooling
- Tiled
- Level loading and switching
- Persistent data, static

During the Lab / This Week

- Implement a workflow for *loading* and *editing* levels / Scenes:
  - Arrays, or
  - Tiled
- Implement a way to *switch* levels (destroy / create)
- Keep your code clean! (Interface vs implementation. Responsibilities. Lifetimes)

Optional:

- Add (player) data that is kept during level changes
- Add a menu / end screen (that resets the data?)

Possible Next Steps

- Our level loading code can still cause bugs if we're not careful! → more details next week
- Using the Tiled *MapParser* class you can customize the loading procedure (e.g. load wall, platform, lethal, climbable tiles from tile layers) → See the tutorial video on Blackboard
- To respawn the level (when the player dies), we *reload* it from file. This is *inefficient* → you can add *respawn* functionality to the Level class (though you need to do this carefully to prevent bugs!)

Next Week

- Some loose ends (more robust level loading, events)
- User interface
- More on inheritance