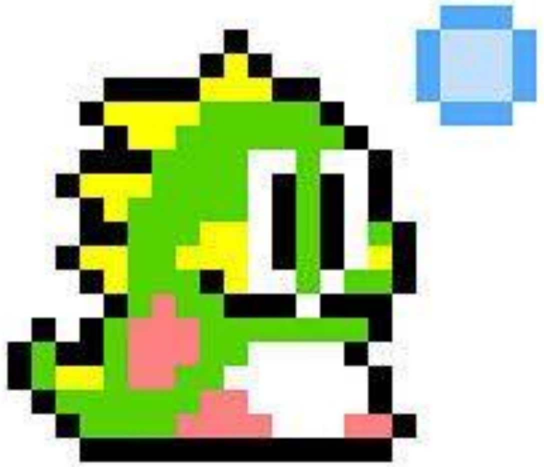
Game Programming

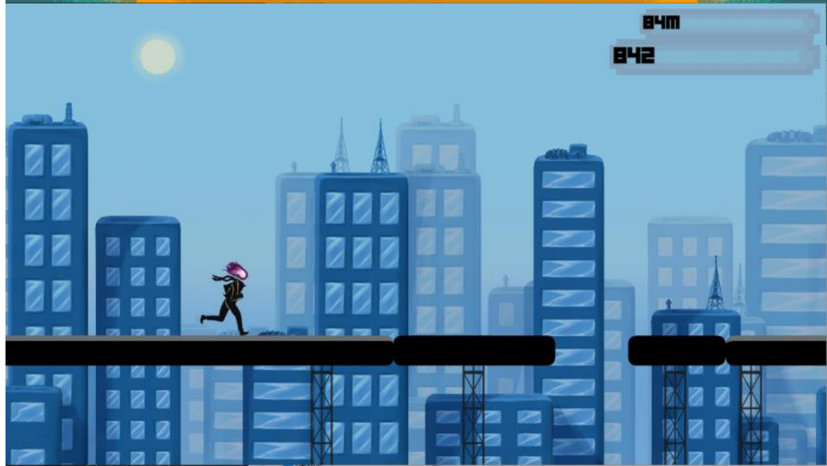
Lecture 1: Introduction + Getting Started

December 4, 2023

Paul Bonsma



Course Goals



This is where you'll learn to create simple 2D games

High level goals:

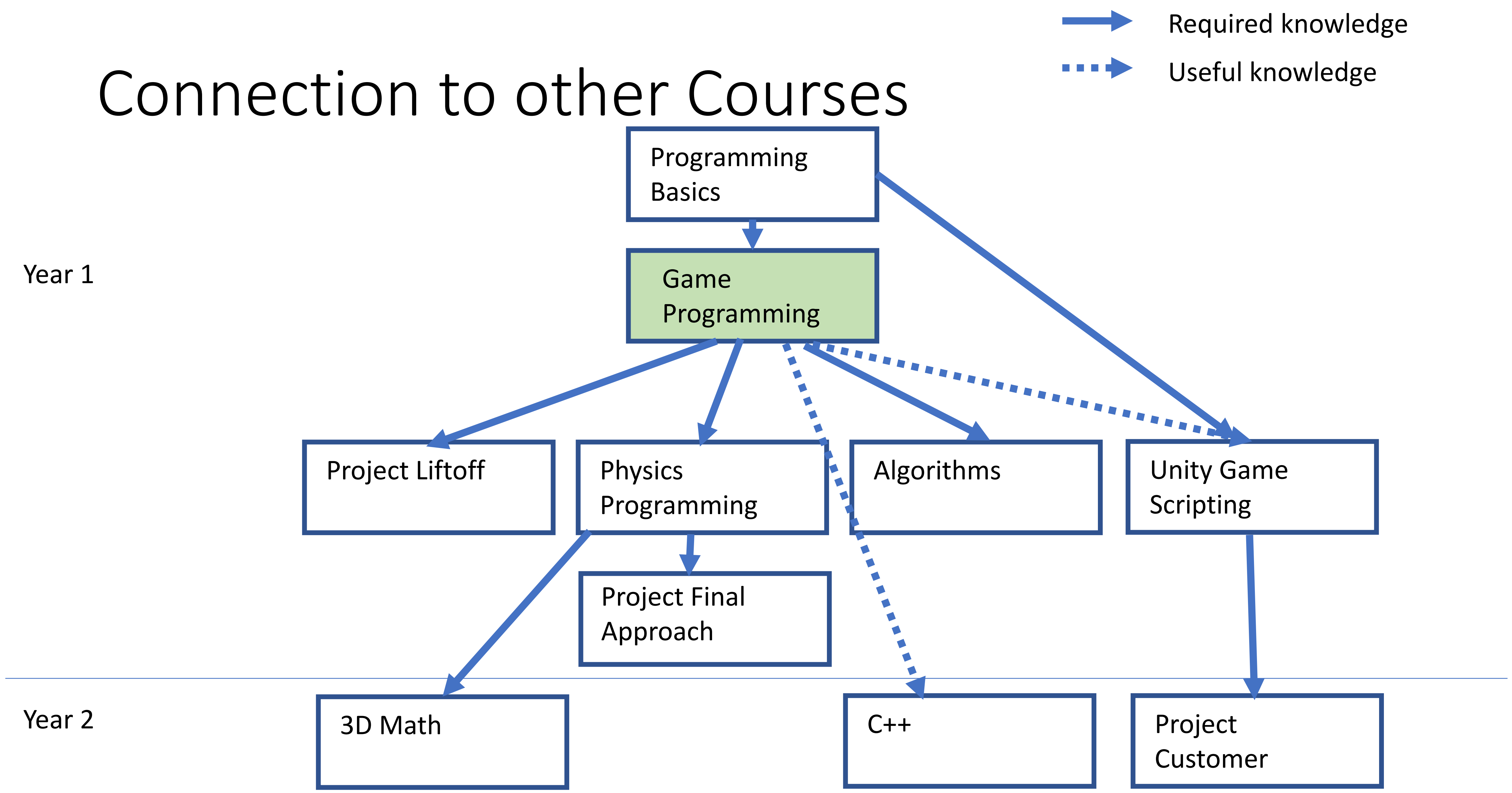
- Preparing for the upcoming *projects* and *courses*
- Improving (Object Oriented) Programming skills
- Learning a simple but fun 2D game engine: GXPEngine (code based, C# language, open source)

Project 1.3 Lift Off

- *Artist*: create assets (art, animation, sound), develop look
- *Designer*: iterate on game design, level design (design, implement, play test)
- *Engineer*: make it work; provide *tooling* to enable the rest of the team

- If you make good tooling, the rest of the team can do more and take work off your hands! :-D
- If you don't provide good tooling, you'll end up implementing everything yourself (typically, shortly before the deadline) :-/
- Make sure you are prepared for the project!

Connection to other Courses



Required Knowledge

- Required: 1.1 Programming Basics (Variables, types, expressions, control structures and loops, methods, passing values and references, return types, variable scope, classes, arrays, …)

Poll:

1. Some of those terms are new to me?
2. Some of those concepts are still a bit fuzzy for me
3. I aced the Programming Basics exam
4. Other
5. I'm just clicking random buttons here

Java vs C#

- "...But we learned Java, and now we'll use C#!"

→ From where you're standing, these languages are almost exactly the same

→ (and where they differ, C# is better ;-))

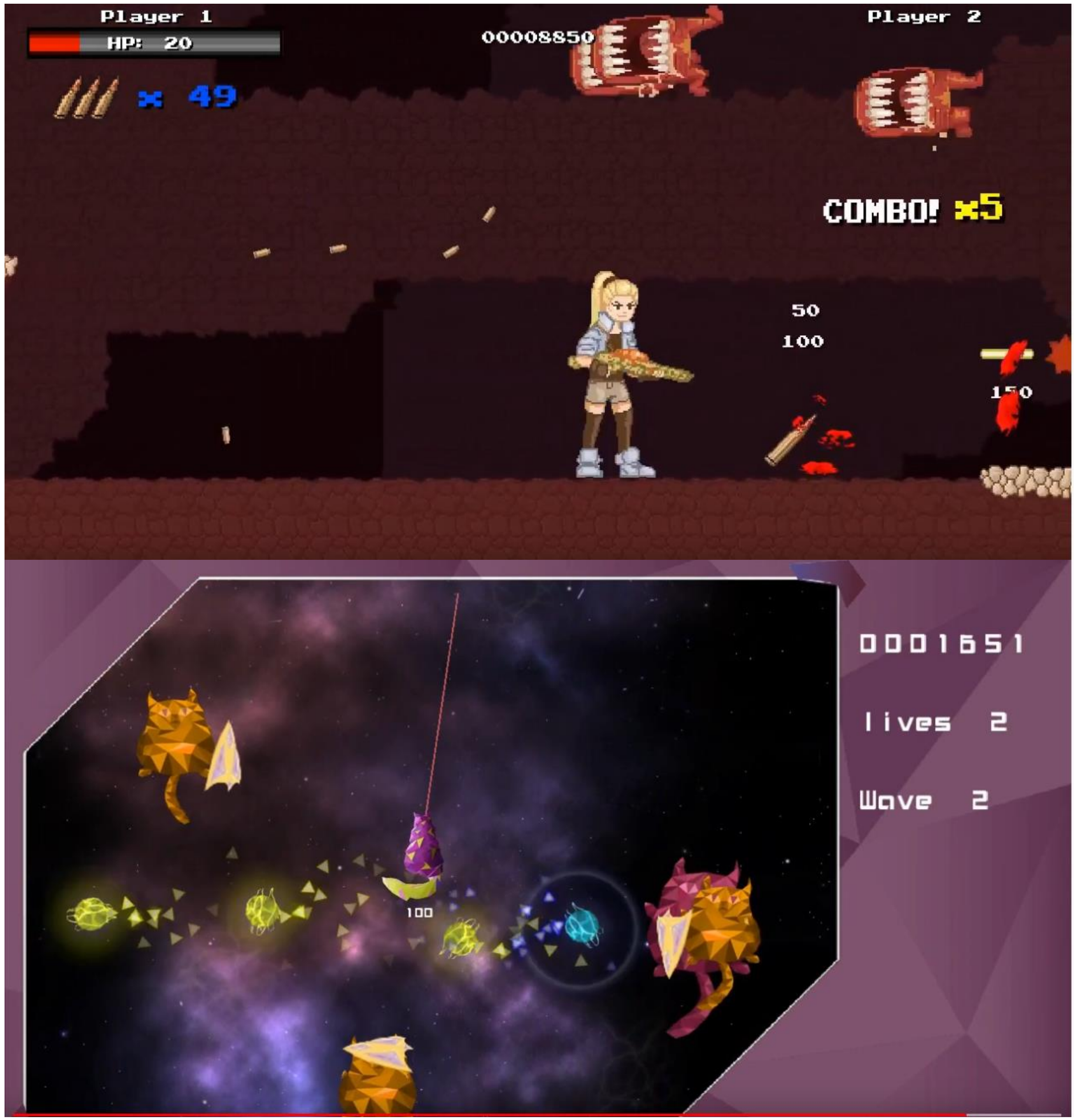
Java	C#	Namespace:
println("Hello world!");	Console.WriteLine("Hello world!");	System
String comparisons don't work :-/	String comparisons work :-D	
ArrayList<SomeType>	List<SomeType>	System.Collections.Generic
class Subclass extends Superclass	class Subclass : Superclass	

Finding more information: *docs.microsoft.com* → google will point the way: just type e.g. *List C#* in google

GXPEngine

- Sprite based, 2D
- Code based environment
- In-house CMGT engine
- *Tiled* as level editor
- API somewhat similar to Unity

- Show reel: https://youtu.be/2wlD0vNPFjM



Poll

How prepared are you?

1. Um… I’m here now, so it could be worse, right?
2. I read parts of the manual
3. I did install Visual Studio Community (or another IDE) + .NET Framework (or Mono)
4. I downloaded, opened and ran one of the projects from Blackboard
5. I already experimented a bit and changed or added some code

GXPEngine vs Processing

Processing

- Good for multimedia (webcam, video, image processing, …)
- Hard to make games with

GXPEngine

- No multimedia (just 2D graphics and sound)
- Good for making 2D games quickly (basic physics, game object hierarchy, sprite animation, level loading, …)

GXPEngine vs Unity

Unity

- Makes many complex things possible, but…
- …makes some easy things hard
- Hidden details (sometimes confusing)
- Good documentation
- Lots of tutorials, *ready to be copy-and-pasted*
- Designer oriented
- 2D and 3D

GXPEngine

- Limited, but…
- …easy and fast to use within those limits
- Open source, extensible
- You have all the code (=better than documentation??)
- You have the opportunity to *solve problems* yourself → the only way to *learn programming*
- Programmer oriented
- 2D only
- Feel free to modify / customize / extend it!

GXPEngine

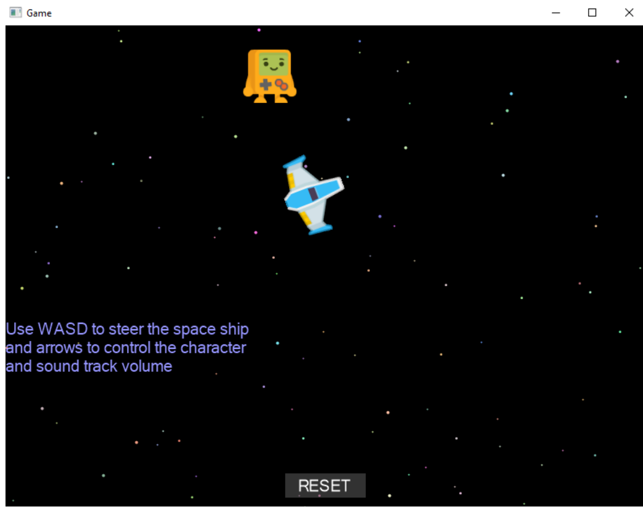
- If you know Object Oriented Programming and some basics of the engine, the GXPEngine is a quick and fun way to make “arcade like” 2D games

- In this course, we will address that “if”

- Along the way, you’ll
  - solve many small and large programming problems
  - Learn several game programming tricks and concepts

Hello GXP (vs OOP)

- On Blackboard, you can find the "Hello GXP" demo project. It shows nearly all the basics (drawing sprites, animation, drawing text, playing sounds and music, checking collisions, getting keyboard and mouse input, changing colors, changing position, rotation and scale)



Use WASD to steer the space ship and arrows to control the character and sound track volume

- …so we're done here?
- No: the main challenge in programming is *managing complexity*
- If you powered through Project First Contact with *many global variables, one huge update function*, you now know what I'm talking about…

→ *Object Oriented Programming* to the rescue!

Poll

My code in Project First Contact...:

1. What? You tell me I could have used multiple files, and non-global variables??
2. I Got It Done, but learned that I should set it up differently next time…
3. I used some classes, and it helped
4. Utility classes, inheritance, no code repetition - OOP FTW!
5. I used so many abstract classes and interfaces, it was hard to find a line of code that actually did something

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

Debugging, problem solving, optimization

Engine knowledge, game programming tricks

Some design skills

General programming skills (managing complexity!)

Project preparation, workflow

Course Outline

1. Introduction and getting started
2. Game objects and collisions
3. Levels and scenes
4. User interface
5. VFX / SFX
6. Advanced topics

Assessment

- Make a 2D game in the GXPEngine
- During the *assessment*, show it, explain your code and choices
- Good code quality required
- Required features and rubrics: *see manual*

- Suggested inspiration: *simple 80s or 90s games*, e.g. Frogger, Tapper, Arkanoid, Bubble Bobble, Donkey Kong, Micro Machines, Boulder Dash…
- You can design your own game, or try to recreate an older game
- Discuss with your lab teacher whether your idea is feasible!

Lecture Outline

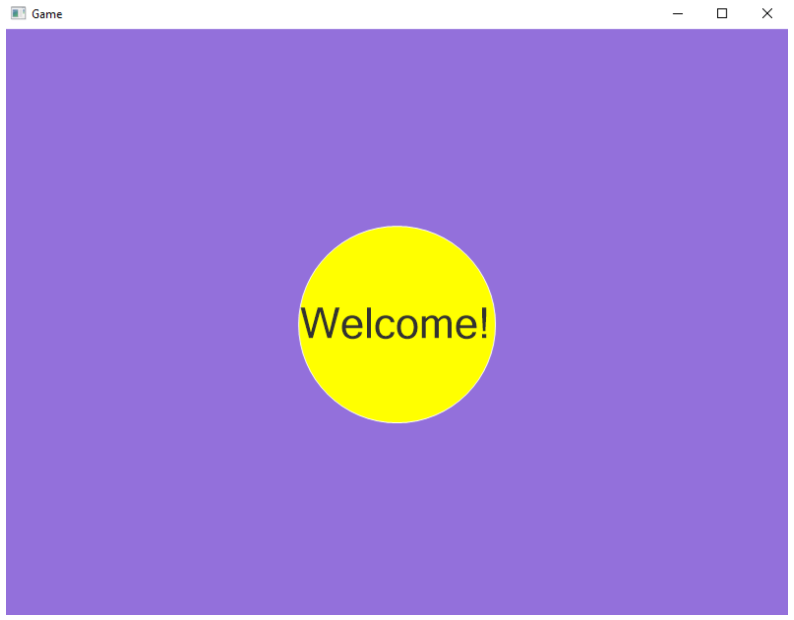
- Getting started with C# and Visual Studio
- Sprites
- Classes & inheritance
- Position, rotation, scale
- Keyboard input
- Sprite animation
- Next steps / getting more information

Getting started

- Install an *IDE* such as **Visual Studio Community**, together with
- **.NET Desktop development** workload (.NET = *runtime)*
- Mac: you may need to install *Mono* as runtime

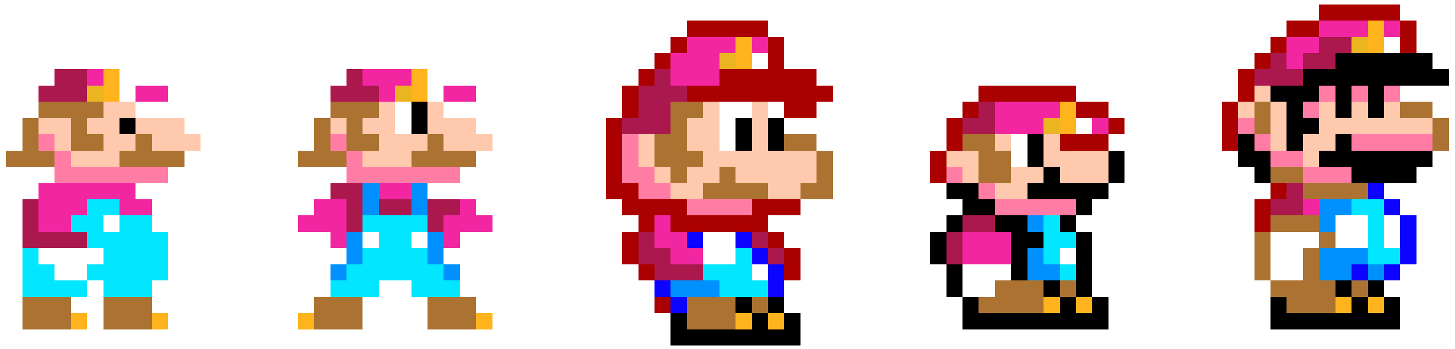
- Download the *GXPEngine* from Blackboard
- Open the *solution file* (.sln) with your IDE
- Click Start (or Build, or Run), or press F5 →

- Ask your lab teacher if you're having trouble
- Open the *solution explorer* to see the engine files



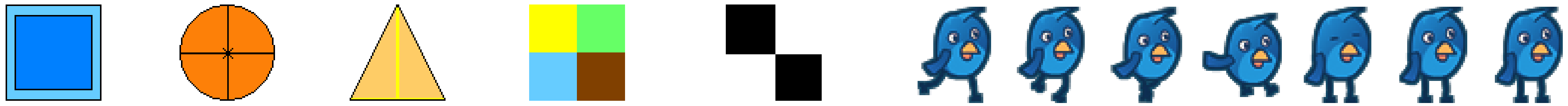
tadaa!

Sprites



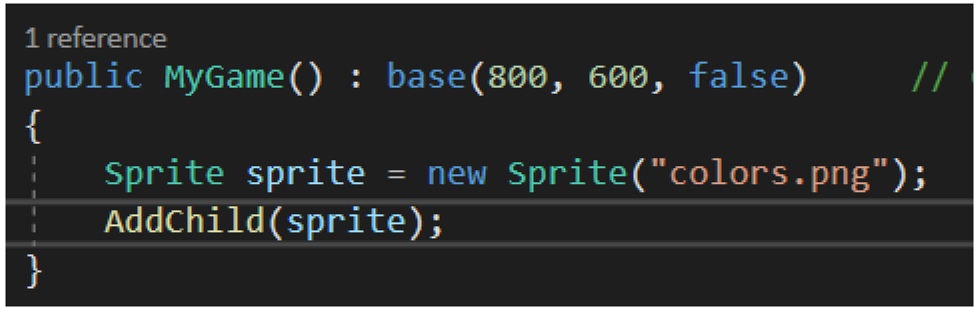
- *Sprites* are (rectangular) game objects showing an image
- These are loaded from image files (typically .png)
- These image files should be in the *bin/Debug* folder
- A few simple (placeholder) images are already included

- Sources for more (free) assets: www.opengameart.org www.kenney.nl www.itch.io etc.



Adding a Sprite

- *new Sprite(imagefilename)* creates a new sprite
- *AddChild* adds a sprite to the engine → it will be displayed
- This is how it's done in HelloGXP



```
1 reference
public MyGame() : base(800, 600, false)     //
{
    Sprite sprite = new Sprite("colors.png");
    AddChild(sprite);
}
```

- It's better to create a *class* for each type of game object / sprite (e.g. Player class, Enemy class)
- This class should *inherit* from the *Sprite* class

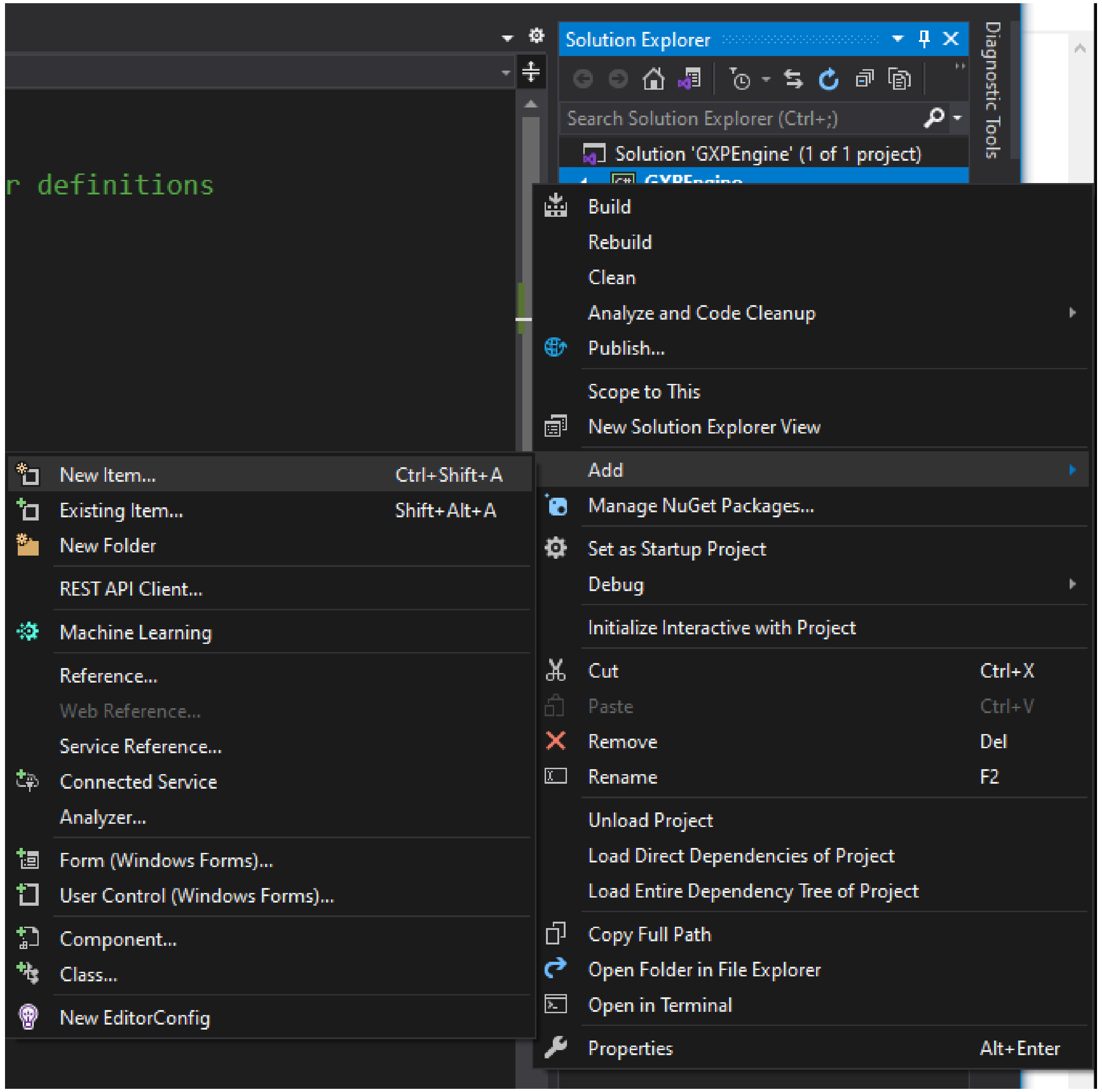
Classes, Instances, Inheritance - Recap

- A *class* corresponds to a type of object
- You can create multiple *instances* of a class, all with their own data *values*
- The class determines the data *types* and *functionality* of the instance
- For example: sprites with different positions
- Using the *new* keyword, you *instantiate* a class (=create an object/instance)

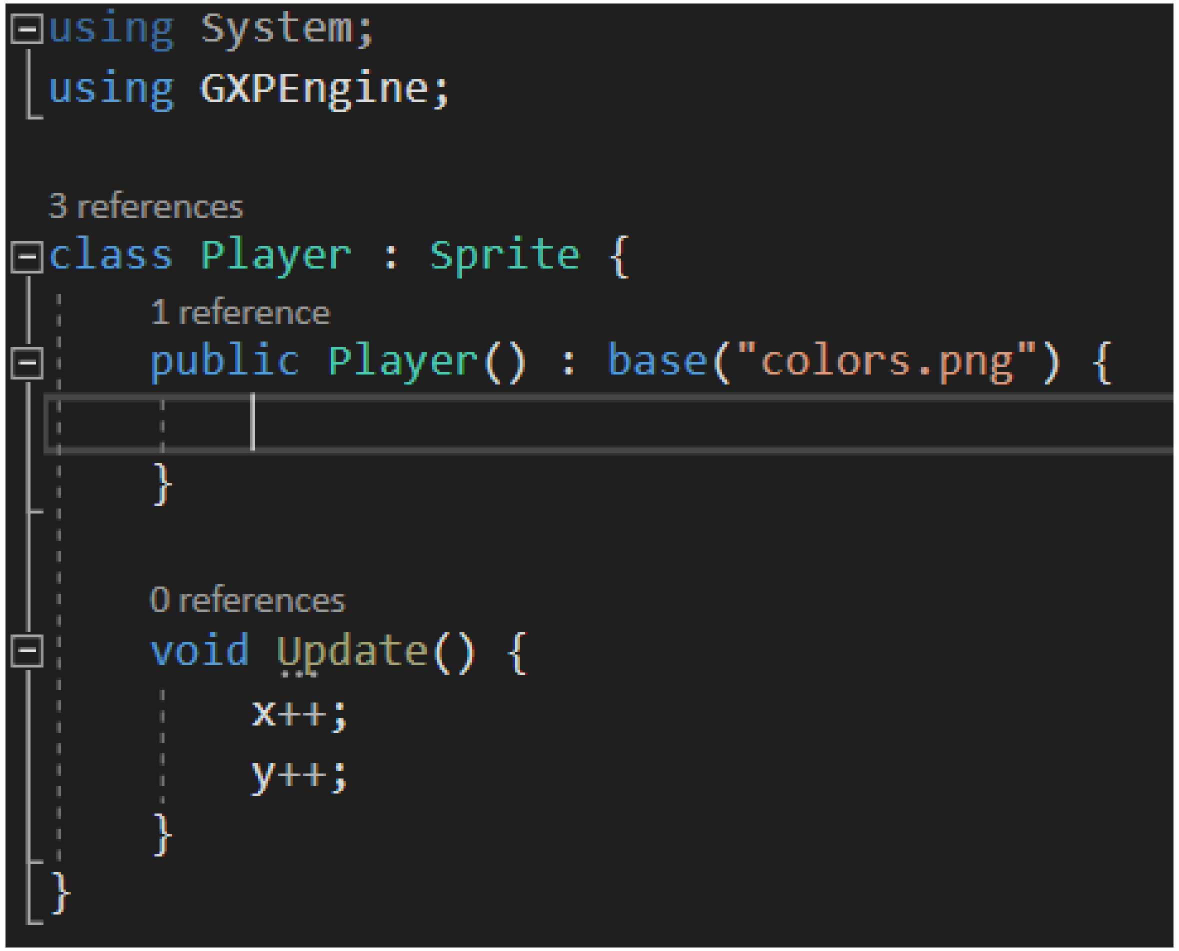
- You can *inherit* from another class (your *superclass* or *base class)*, to get all of the data and functionality of that class for free
- We also say that the subclass *extends* the superclass
- Is this all clear…?   (Terminology = important!!)

Creating a Player Class

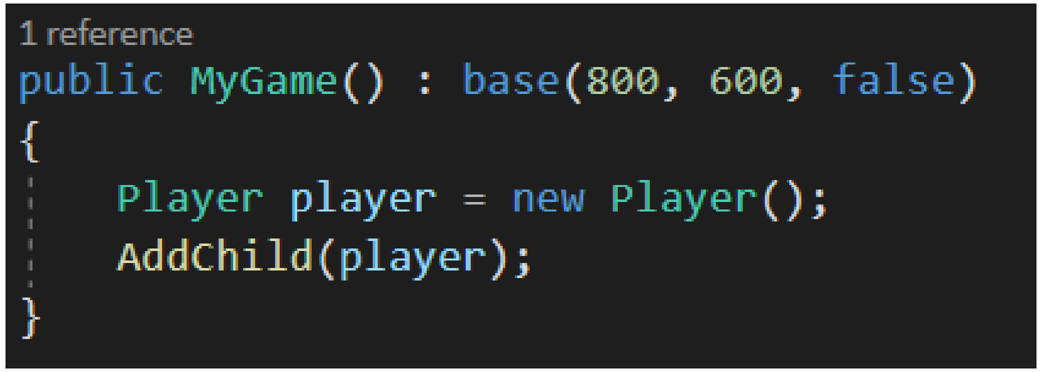
- In the solution explorer, right click the project (GXPEngine),
- ...select Add → New item → Class
- ...and call it Player.cs



Player Class



```
using System;
using GXPEngine;

3 references
class Player : Sprite {
    1 reference
    public Player() : base("colors.png") {
        
    }

    0 references
    void Update() {
        x++;
        y++;
    }
}
```



```
1 reference
public MyGame() : base(800, 600, false)
{
    Player player = new Player();
    AddChild(player);
}
```

Player Class

Note:

- *x* and *y* (part of Sprite class) control the position in pixels
- Origin (0,0) is top left
- *Update* is called by the engine, every frame, for every Sprite that is added with AddChild (typically 60 FPS) (it's like *draw* in Processing)
- With "*base*" you can call the superclass constructor of the Sprite class (necessary!)
- Adding "*using GXPEngine*" is necessary to use the engine's classes

Naming Conventions

- Classes *always* use *PascalCase*
- Methods should use *PascalCase*
- Properties, fields and variables use *camelCase*

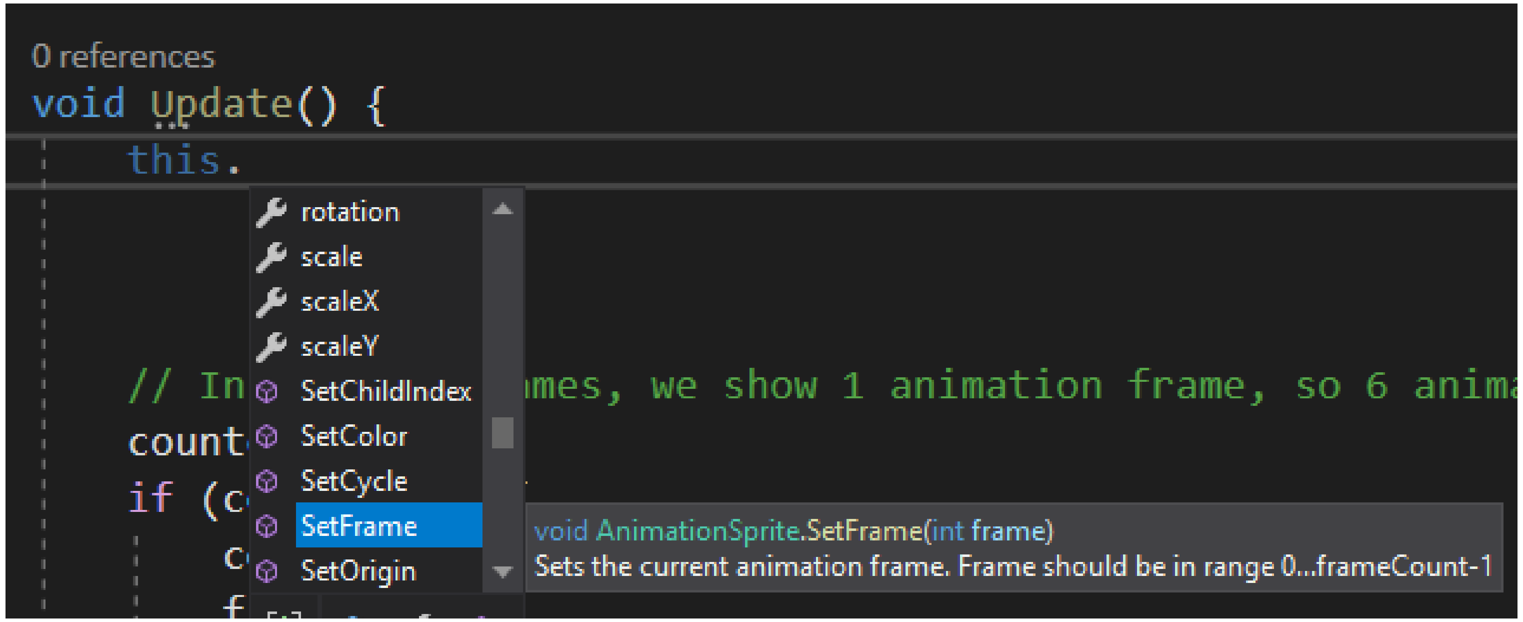
- (The engine itself still uses some older code conventions, which are kept for compatibility… Sorry about that…)

Creating Multiple Sprites (Demo)

- We can easily create multiple instances of our Player class
- Use *Utils.Random(min,max)* to get random numbers (from min to max-1!) → random positions

- …what else can we do with our player?

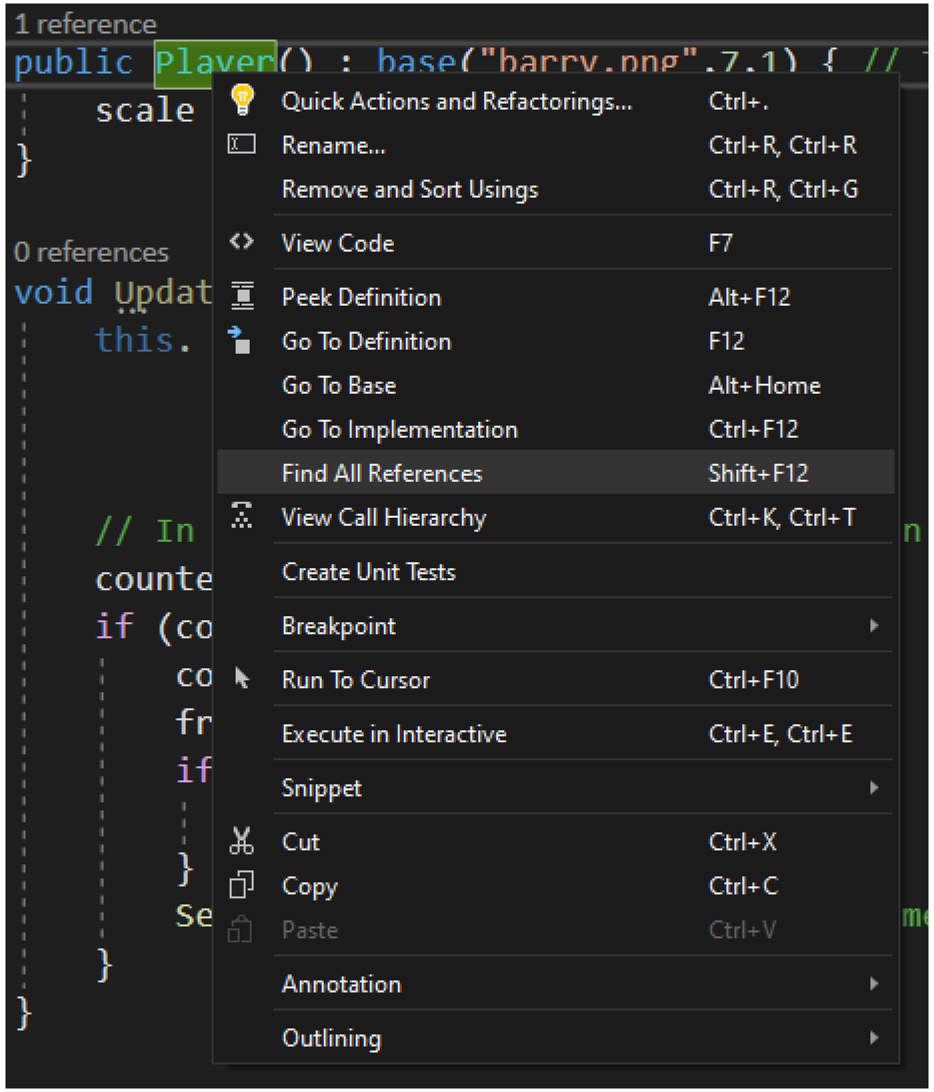
Exploring the engine (this., F12, comments)



```
0 references
void Update() {
    this.

    // In               mes, we show 1 animation frame, so 6 anima
    count
    if (c
        c
```

void AnimationSprite.SetFrame(int frame)
Sets the current animation frame. Frame should be in range 0...frameCount-1

- Visual Studio is your friend ☺


Quick Actions and Refactorings... Ctrl+.
Rename... Ctrl+R, Ctrl+R
Remove and Sort Usings Ctrl+R, Ctrl+G
View Code F7
Peek Definition Alt+F12
Go To Definition F12
Go To Base Alt+Home
Go To Implementation Ctrl+F12
Find All References Shift+F12
View Call Hierarchy Ctrl+K, Ctrl+T
Create Unit Tests
Breakpoint
Run To Cursor Ctrl+F10
Execute in Interactive Ctrl+E, Ctrl+E
Snippet
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Annotation
Outlining

```
1 reference
public Player() : base("barry.png",7,1) { //
    scale
}

0 references
void Updat
    this.

    // In
    counte
    if (co
        co
        fr
        if

        }
        Se
    }
}
```

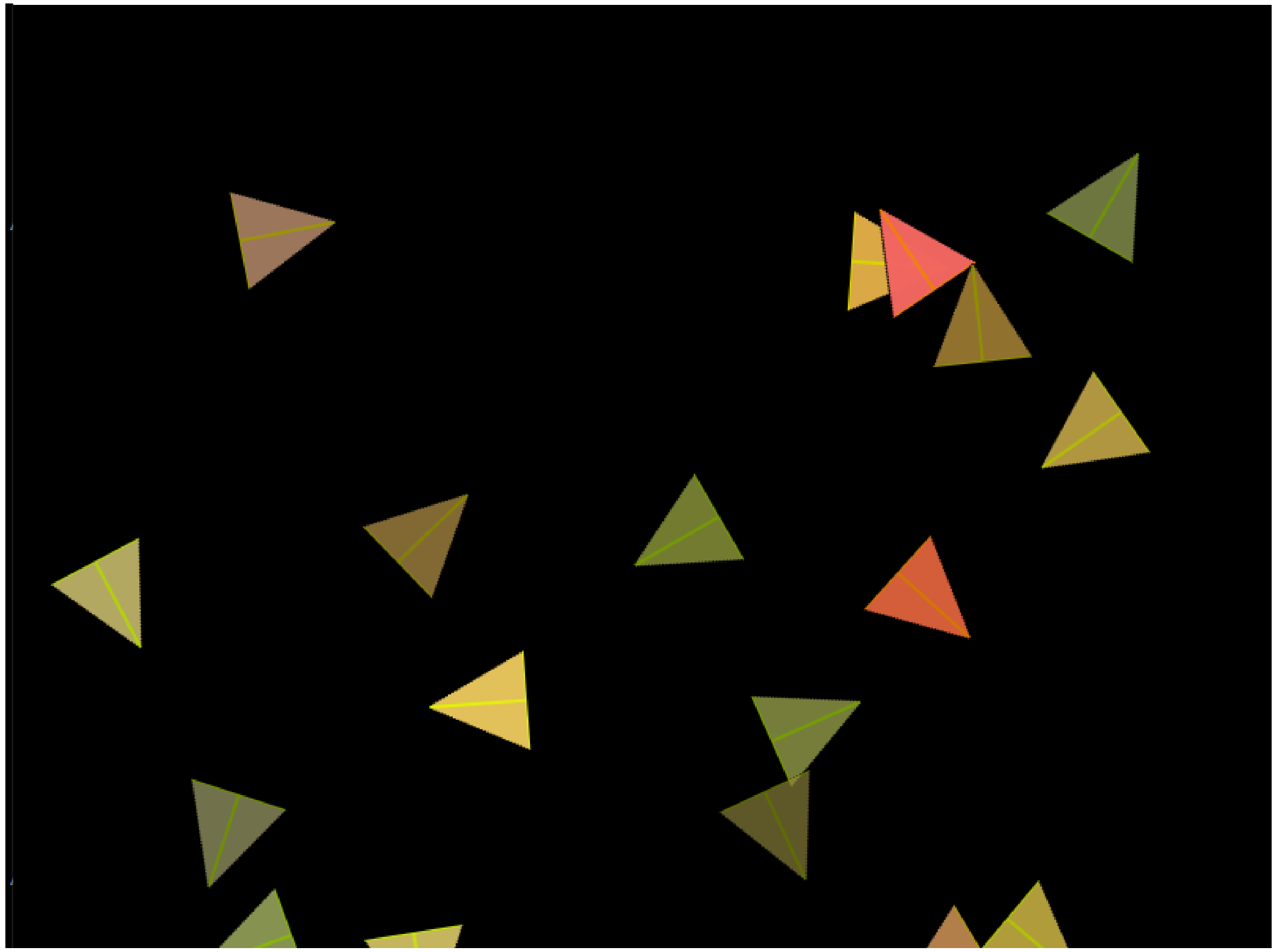
More Fun with Sprites (demo)

- x,y
- rotation
- scale
- Move
- Turn
- SetXY
- width, height
- SetOrigin
- SetColor
- alpha
- Mirror

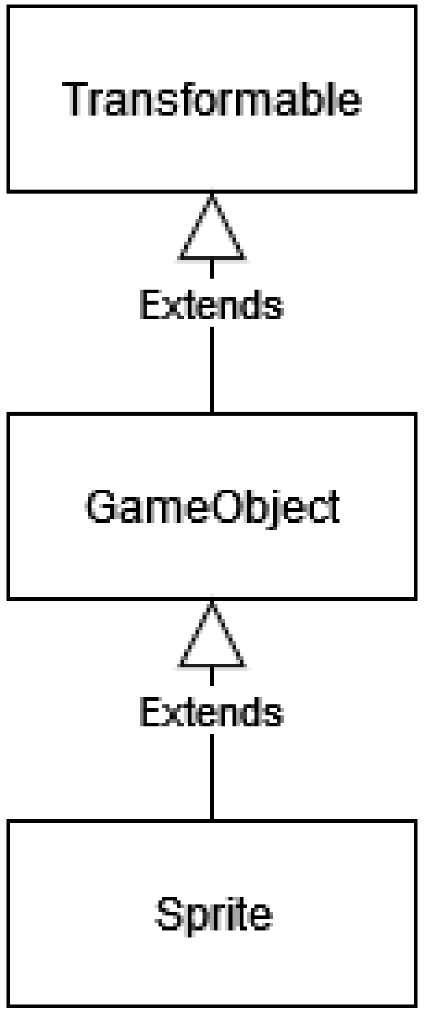
Transformable

Sprite

…all part of the Sprite class? No!



Inheritance used by the engine



position, rotation, scale

parent / child (AddChild) → more details next week!

Image (texture), color, alpha, mirror, width, height

Player Input (Demo)

- Use *Input.GetKey*, *Input.GetKeyDown*, *Input.GetKeyUp* to get keyboard input
- Key codes are defined in the *Key* class (e.g. Key.LEFT, Key.SPACE, …)
- Check out Key.cs (in the *Utils* folder) for the full list!

- Result: a controllable player

Console & Debugging

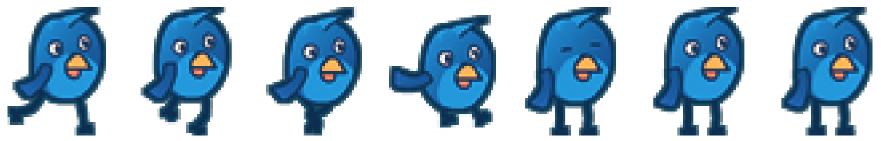
- Simple debugging (for now):
- With *Console.WriteLine* you can write information (variable values) to the console
- *PRO TIP:* in Visual Studio, type cw and then tab twice!
- The console is hidden behind the game window

- If you do this in update: use Input.GetKey to prevent spamming the console!

Code Quality - 1

- *DRY: Don’t Repeat Yourself!*
  → Repetitive code is *unmaintainable* code!
- *Avoid hard coded values!*
  → Every number is a parameter a game designer may want to tweak
- *Keep methods short*
  →Split them up into pieces that logically belong together
- *Keep variable scope small*
  →Use local variables (in methods) instead of class-level variables *(fields)* unless you have a good reason for it

Sprite Animation



7 columns, 1 row

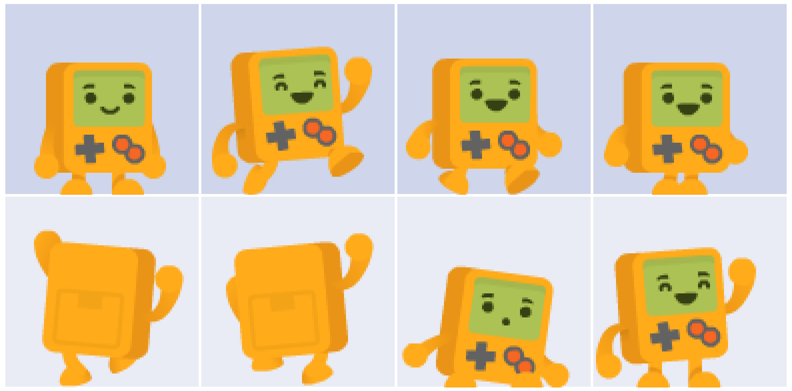
- Animating a sprite requires a *sprite sheet*
- The GXPEngine requires the animation frames to be spread out in a *regular* grid (every frame has the same size)

GOOD

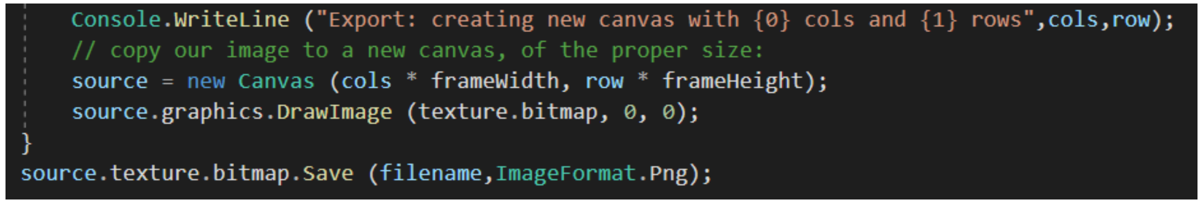
BAD:

columns, rows??

- You might find some sprite animations that consist of single images, or *irregularly* placed
- There are apps to fix that, such as
  - https://renderhjs.net/shoebox/
  - *(Expert option:)* You can write it yourself :-D

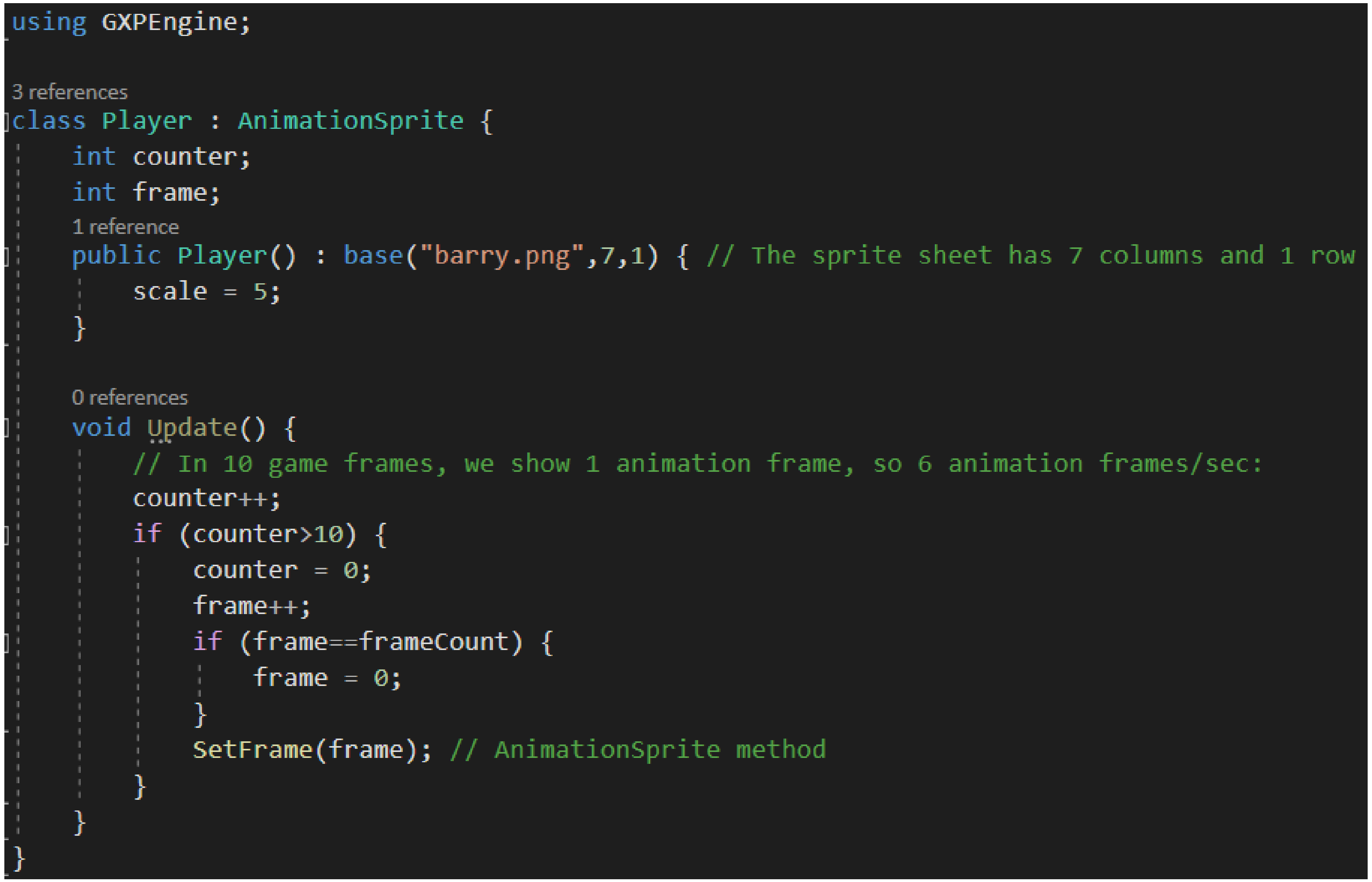


4 columns, 2 rows



```
    Console.WriteLine ("Export: creating new canvas with {0} cols and {1} rows",cols,row);
    // copy our image to a new canvas, of the proper size:
    source = new Canvas (cols * frameWidth, row * frameHeight);
    source.graphics.DrawImage (texture.bitmap, 0, 0);
}
source.texture.bitmap.Save (filename,ImageFormat.Png);
```

AnimationSprite



```
using GXPEngine;

3 references
class Player : AnimationSprite {
    int counter;
    int frame;
    1 reference
    public Player() : base("barry.png",7,1) { // The sprite sheet has 7 columns and 1 row
        scale = 5;
    }

    0 references
    void Update() {
        // In 10 game frames, we show 1 animation frame, so 6 animation frames/sec:
        counter++;
        if (counter>10) {
            counter = 0;
            frame++;
            if (frame==frameCount) {
                frame = 0;
            }
            SetFrame(frame); // AnimationSprite method
        }
    }
}
```

AnimationSprite methods (Demo)

- *SetFrame:* Select a frame

- *SetCycle*: set the animation range (if e.g. walk, idle, jump are part of the same sprite sheet)
- *Animate*: automatically animate within the chosen range

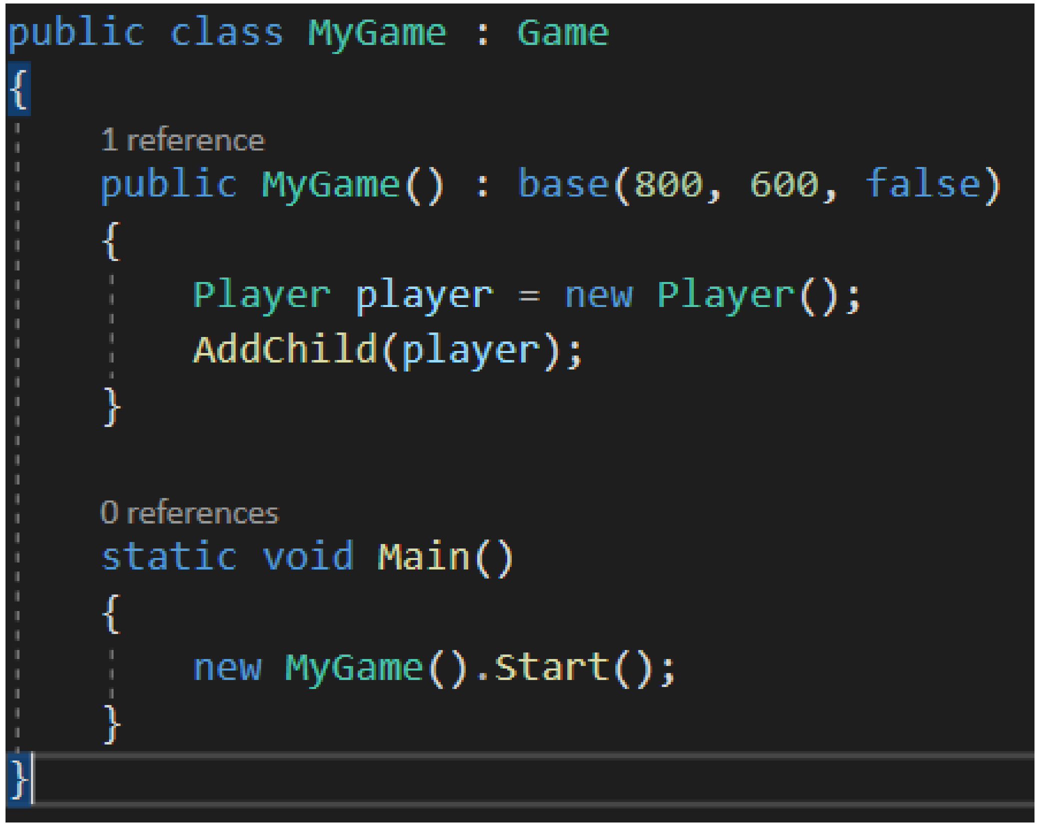
- See also HelloGXP

Game, main

- Your MyGame class should inherit from *Game* (which is also a GameObject)
- There can only be one Game instance!
- In the Game constructor, you can set the screen resolution and whether it should be full screen or not

→ For now, work without full screen!

- *static void Main()* is the method where the program starts: create your one MyGame instance and start it



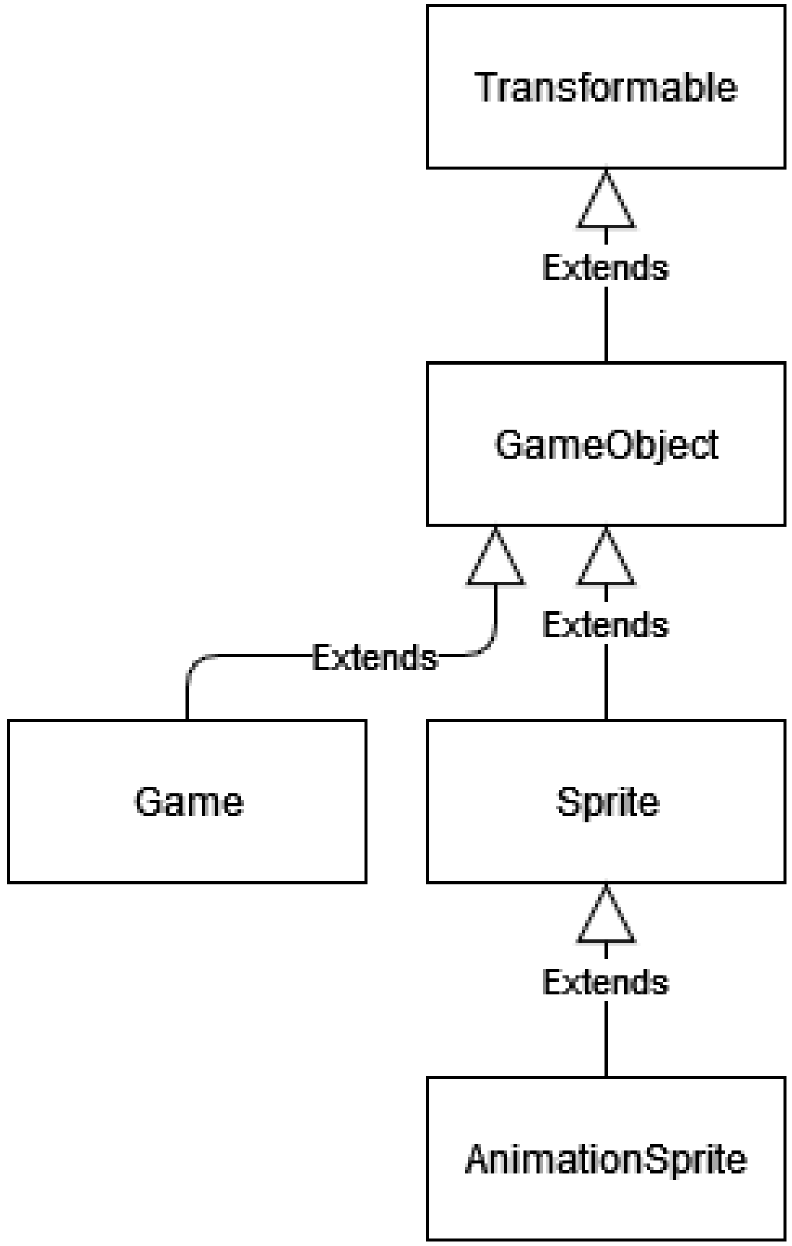
```
public class MyGame : Game
{
    1 reference
    public MyGame() : base(800, 600, false)
    {
        Player player = new Player();
        AddChild(player);
    }

    0 references
    static void Main()
    {
        new MyGame().Start();
    }
}
```

Summary

This week we have seen:

- Transformable (position, rotation, scale)
- Sprite (width, height, color, alpha, mirror)
- AnimationSprite (SetFrame, etc.)
- Input (GetKey, etc.)
- GameObject: AddChild
- Game
- Utils.Random

→ Practice with all of these things this week!



During the labs

- Create an animated player object that can be moved using keyboard input
- Practice with all methods / variables shown today

Optional:

- Just play around with this – you might just stumble upon some *playful game mechanics!* (For example: add some chase/evade/capture goal to today's demo and you already have the core of a game!)
- Find some nice sprites / sprite sheets online

- Note: 3ec = 82 hour ≈ 13 hour per week → practice outside the labs / lectures!

How to get more information?

GXPEngine: (see Blackboard)

- HelloGXP
- Cheat sheet
- Full documentation? (if you dare…)

- Just look at the engine code! (Start with the *root folder* and *Utils*)
- *Best tip ever:* this. F12 comments

C# / general:

- docs.microsoft.com
- google

- Go to the labs and ask questions! (We have 3 experts, eager to help!)

General Studying Tips

- Make *backups* before you break things again! (Using *git* might help)
- Use the *cheat sheet* from Blackboard: extend it, or make your own
- Set *small and achievable goals!* (First Pong, then Breakout, then Mario)
- *Reflect* on your code, and *experiment* with it
  - E.g. reorder / comment out parts
  - What does each line do? How does the order matter?
  - Can you get the same functionality in a better (more maintainable) way?
- Don't *stay stuck* for too long
  - In the *labs* and in the *CMGT Discord*, we're eager to help
- It's good to form *study groups* and *learn* from each other
  - …but not to just *copy-paste* from each other!
  - Give *tips* and *feedback*, but let them write their own code

Next Week

- Game Objects: *creating* and *destroying* them dynamically
- *Collisions*, and creating *interactions* between different objects