University of Szeged

Faculty of Science and Informatics

Bachelor’s thesis

Lajos Rajna

2021

University of Szeged
Faculty of Science and Informatics

Development of a cross-platform 2D video game engine using MonoGame framework

Bachelor's thesis

Author:
Lajos Rajna
Computer Science student

Advisor:
Dr. Peter Bodnar
Assistant professor

Szeged
2021

Goal setting

The goal of this thesis is to cover the basics of 2D video game engine creation, without any platform-specific code or solutions. The engine will utilize the MonoGame framework, which was initially an open-source reimplementation of Microsoft’s XNA game development framework, but has evolved since then and became a platform-independent 2D and 3D library, providing the basic functionalities for cross-platform video game development with an MIT license.

Summary

- ***Project name:***

Creation of a free, cross-platform, open-source, 2D video game engine. The student's task is to create and demonstrate a working video game engine fulfilling the above criteria.

- ***Description of the task:***

Although there are numerous commercial video game engines available, all of them lack at least one of the four criteria: free, open-source, cross-platform, and has good 2D performance. The task is to create a video game engine that fulfills these criteria and provides the most common functionalities needed to create video games.

- ***Solution:***

Creation of the above mention video game engine using MonoGame framework and C# language. The MonoGame framework provides low-level, cross-platform functionalities while C# will allow the creation of the engine's components.

- ***Tools and methods:***

C# languange and MonoGame framework.

- ***Results:***

A fully functional engine has been implemented that encapsulates the most common functionalities that 2D video games need and satisfies all four initial criteria. Furthermore, a short 2D platformer video game sample is included in the project to demonstrate the engine's capabilities.

- ***Keywords:***

MonoGame, C#, 2D, performance, cross-platform, open-source, video game engine

Table of contents

Terminology

- **Indie**: an independent video game developer entity. It can be one person or a small team, usually not funded by any major company.
- **Video game engine**: the source code responsible for running the game including rendering, audio, physics (if applicable), inputs, and any other code piece that somehow contributes to the game experience. It can be a generic, commercial engine or a smaller piece of code written to run one specific game.
- **Sprite**: a 2D image drawn on the screen
- **Asset**: a 2D image, 2D/3D model, animations, music, scripts, visual and audio effects, artworks, fonts, and any other digital product that a video game contains
- **Axis aligned**: an N-dimensional object whose shape is aligned with the coordinate axes of the space.
- **SpriteBatch**: a group (or batch) of 2D images that will be drawn with the same settings by the GPU
- **AAA video game**: video games developed with a high budget, usually involves a bigger team that includes engineers, artist, musicians, animators, etc., published by a big company
- **FPS**: frame/second, the number of frames that the hardware produces in one second
- **NPC**: non-player character: friendly video game characters controlled by an AI (like citizens of a town, etc.)
- **Platformer game**: also commonly referred to as "side-scroller" game, a video game where the character can move around in the 2D space, typically moving from left to right using jumping and other functionalities to complete the levels. The most commonly known example is the Super Mario Brothers.

Notations

- `Consolas font:` source code snippets
- *Italic:* Method, class, or variable names embedded in the text, referring to the source code or an example code snippet
- *.cs: every C# source code file in the directory

1. Motivation, solution, and tools

In the early days of video game consoles and personal computers, video game developers were mostly solo developers, or small teams, working a video game in their free time or using their savings or mortgages to fund the development costs. Once video games started to become popular in the next decades, the industry was taken over by big publishers who had the funds and the knowledge base to create high-quality video games with budgets that indies never had. By the early 2000s, almost all popular video games were created by an established publisher. Engines for video games were either written in-house by each studio, or were licensed from another major company for hundreds of thousands, or even millions of dollars. Independent video games slowly got suppressed out of the video game landscape and AAA titles took over the market.

By around 2010, the situation had started to shift in favor of independent developers again. Technology companies have realized that making video game engines publicly accessible for a small fee is much more lucrative than just selling the license for a few other companies. The technology and programming languages have also evolved to a level where it was much easier to make video games again by one person or a small team, development kits for video game consoles became affordable, and having high-speed internet in most households also contributed to the fact that small teams did not need to be backed anymore by big publishers to distribute the products on CDs and DVDs, but they could just release their game fully digitally in online stores and players would download the purchased games. Digital markets are becoming bigger and bigger these days, and although optical discs are still present and have a major share in sales, digital downloads have now taken over in the number of overall sales and are expected to continue their growth in the future as well.

One more factor that might have contributed to players turning their attention to independent video game development was the diversity of games. Major developer companies usually stay on the safe side and only release games that are highly likely to become a financial success, whereas in many cases, independent developers tend to take a much higher risk to pursue a vision to create something original. It was not uncommon that a game idea was rejected by different publishers, but the development got funded via crowdfunding and became a huge success after release. This takes us to the last point contributing to the new age of independent game development: the publicity and general trust towards crowdfunding. Lots of video games get funded via crowdfunding websites like Kickstarter and IndieGoGo. In general, people seem to appreciate the effort put into video games and are willing to take the risk of supporting a video game that is still under development. This opened up the possibility not just for creating small games, but in many cases, huge funds are donated towards a popular game, much more than the developers originally asked for and they deliver an AAA quality product without the help and influence of a publisher.

My goal is to create a cross-platform engine that has good 2D performance on low-end hardware (older smartphones), open-source, and completely free to use.
When looking at the commercial engines publicly available, although they are great, each engine lacks something of the above-mentioned 4 criteria.

1.1 Solution

Using the MonoGame framework and C# language, I am going to implement a 2D video game engine that would fill the gap in the video game engine landscape by fulfilling the above-mentioned 4 initial criteria: maximized 2D performance and usability, open-source, free and cross-platform.

Challenges I faced during my work:

- To implement the aforementioned functionalities in a platform-independent, effective way with good performance and high reusability
- To create an engine that is easy to understand for newcomers but complex enough to support almost any 2D game.
- To be generic, but still provide the basic functionalities for most genres: It can be challenging to decide what should be part of the engine and what should be coded by the game developer: in general, I did not include any genre-specific code in the engine, I tried to be as generic as possible. The only exception is supporting gravity, which is part of the physics engine and although not all games need it (like top-down RPGs or strategy games), it's still such an important part of most games that I implemented it natively. The game developer can turn it off if it's not needed. In the future, I will provide a *GameUtils*

extension library to implement basic, genre-specific functionalities that can be used modularly, but not part of the engine's core itself (like 2D pathfinding, shaders, etc).

- To decide what to support natively: for example complete, realistic physics are extremely hard to implement and are not needed by most 2D games at all. But there are open source, third-party libraries available to provide such functionality, the developers can use these with the engine as needed. But a basic, axis-aligned physics is included that is sufficient for most 2D games.
- To find the balance between high-level development workflow and low-level control: the reason why most people use game engines is that they would not like to "reinvent the wheel", they would like to code on a high level focusing on the game itself and avoid any low-level code that is responsible for rendering, physics, etc. On the other hand, developers that would like to have access to such low-level operations for any reason should be allowed to: while it is completely possible to create a game in the engine without using low-level code, it is always possible to override methods responsible for low-level operations and extend/reimplement them or parameterize classes in an advanced way to gain a finer level of control.

1.2 Tools used to achieve the goal:

1.2.1 C#

C# hardly needs an introduction for anyone with software engineering knowledge: it's a multi-purpose, multi-paradigm programming language, part of Microsoft .Net initiative to create programming languages that run on multiple platforms. This was achieved by Mono: a free and open-source project that allows the development of cross-platform compilers and runtime environments for the language. C# is a managed language (uses garbage collector) that offers Java-like high-level programming while simultaneously allowing low-level operations, like out parameter types (reference passing) and pointers (if the unsafe mode is allowed), therefore a perfect candidate for a video game engine programming language: easy to understand for beginners while low-level enough for professionals.

1.2.2 MonoGame

In 2004, Microsoft announced their game developer toolset called XNA targeting Windows and Xbox 360 platforms, the first public build was released in 2006. The framework relied on C# language and provided basic functionalities for video game development, unifying the development pipeline for PC and console: apart from some inevitable platform-specific changes, the games created with XNA ran on both Windows and Xbox 360 platforms. In 2010, Microsoft extended XNA to support Windows Phone as well. XNA became popular between indie developers and video game studios and hundreds of games got published using the framework, the estimated count of the developers using it was over 10.000. In 2013, Microsoft has officially announced the discontinuation of XNA, and although the reasons were never stated, most probably it was because of the following:

- They were not able to reach a target audience big enough. The framework, being very low level, was too difficult for beginners, and for professional developers, porting their games for non-Windows platforms was very expensive.
- The quality and quantity of the commercially available cross-platform game engines started to rise which, and in many cases, proved to be a more cost-effective alternative to XNA, and they were also more beginner-friendly.

However, there was still a significant mass of developers who already had the XNA framework knowledge, games were already under development using it, and even while XNA was still actively developed, in parallel, MonoGame, an open-source reimplementation of the XNA API appeared in 2009. The original goal was to reimplement the API so that games that were written using XNA would work without a change, but would also run on non-Microsoft platforms, and developers also had access to MonoGame's source code in case they wanted to customize it, which was not the case for XNA. After XNA got shut down by Microsoft, the importance of MonoGame has started to grow, as well as the project. Initially, MonoGame only had 2D capabilities, which after years of development, evolved into a fully functional, cross-platform 3D development library that became widely popular and corrected XNA's shortcomings. The former XNA developers were still active, and a vivid community prospered leading to the framework's popularity, especially among independent developers. Hundreds of games have been published using MonoGame on all major platforms, many of them became huge successes and were recognized by both the players and the critics. MonoGame is still actively being developed today and its popularity keeps rising not just because of its capabilities, but because the C# language is a great language for video game development: it’s managed, so the developers don’t need to care about memory management if they don't want to, but also support low-level operations.

Functionalities provided by MonoGame:

- Draws 2D images and renders 3D models on the display (supports both OpenGL and DirectX)
- Offers built-in effects (shader) system called MGFX which runs on all platforms
- Propagates input from the keyboard, mouse, controllers, and other input devices
- Supports playing audio files
- Provides an implementation of the most important mathematical concepts

Functionalities that MonoGame does not provide, and will be the topic of this thesis:

- Camera
- Physics engine
- 2D animations
- Abstraction for video game entities
- Collisions
- AI
- Any video game logic related code
- Support for external map editors
- Texture caching
- Scene management
- UI capabilities
- Particle system
- Caching, pooling

These are the most important components of a 2D video game engine that allows almost any game to be created, therefore I focused on the general, efficient implementation of these. Due to time restrictions, the particle system and object pooling did not get implemented, I chose to postpone these as without them, the engine is still fully functional. Since this project isn’t just my bachelor thesis, but my own product as well, with which I’m planning to release video games, the missing components will be added in the near future.

1.3 General approach for video game engine development

Developing a video game engine is similar to an API development, where the target audience is also professional software engineers or hobby programmers, therefore the code must have certain standards that programs targeting non-IT people don't necessarily have to. For example, developing an average mobile application or a desktop application, in many cases, doesn’t need to be optimized above a certain level as the developer can simply define minimum hardware requirements. In many cases, they are also not that performance-sensitive, therefore optimization above a certain level is considered overengineering and might be a waste of development resources: generally, the users won’t notice the difference between an action (like a button press) taking 0.5ms or 0.05ms. Having a 0.5 response time in an average application may already be achieved by the initial release of the software while optimizing it to 0.05ms might take up enormous resources without any real benefit or business value.
Web API and video game engines are different: for web API, the code must be ready to serve as high volume of requests as possible with high efficiency, and a video game engine must be able to produce frames as frequently as possible. Also, the code must be prepared for high reusability, the engine should not restrict the developer in what they can achieve and should not limit them in any way. The programming techniques described below can be used to achieve high performance and usability.

Avoid duplicated work, use caching instead
Whenever possible, try to store and reuse the result of previous calculations instead of recalculation. Memory is usually not a bottleneck in any gaming platform, especially when developing a 2D game, where the assets are typically much smaller in size compared to a 3D game and takes up much less space in the memory, allowing us to efficiently cache data.

Prefer callbacks instead of polling

Whenever possible, implement callbacks to trigger certain events instead of continuously checking whether events should occur. This is also partially avoiding duplicate work: if component A in the game has calculated collision with component B, let the collision be triggered in both component A and B instead of B checking for the same collision in its own update loop.

Choose the right data structures

Choose the data structure according to your current needs and keep your asymptotic runtime as low as possible. For example, when storing data in a collection that will be used for lookups, it's better to keep it in a Set or Dictionary as the lookup speed is O(log(N)), rather than using lists where the lookup takes O(N). For this to work efficiently, we also have to choose our keys carefully: keys must be hashed and compared quickly to guarantee fast lookup. Also, many times, we have to modify a collection while iterating through it (for example, we are calling the Update loop on our list of objects and a new object gets created and added to the same list of objects). A common mistake to avoid exceptions coming from modifying the list while iterating through it is always creating a copy of the list for the iteration for each *Update* method call and add the new object to the original list. While this works, it has bad performance as the list will be copied in each *Update* call, maybe hundreds of times/second. Instead, let's store the newly created or removed object in a separate list and add/remove these objects once we have finished iterating through our list of objects (and don't forget to clear the list that holds new/deleted objects). As creating/removing objects usually isn't happening in such high volume, it's much faster than continuously copying our list of existing objects.

Preload and cache assets whenever possible

If the memory allows, preload and cache the game assets (graphics, audio, maps, etc) instead of using them on a lazy-load basis. Typically, players prefer waiting longer during the game startup over having to wait for assets to load during gameplay, which disrupts the experience.

Use a fixed update loop and interpolation

Video games need to run several components to provide gameplay, including collision detection, AI, physics, cross-interaction, etc. A modern CPU can do these hundreds or even thousands of times/second, but it's unnecessary. For an average game, doing these calculations 30 times/second is enough, anything more than this has no effect on the gameplay, but puts a high burden on the CPU, which is mostly a problem for mobile phones and other handheld gaming consoles: high CPU utilization leads to battery drain and can heat up the device quickly. Doing the expensive calculations only 30 times/second and calculating the rest of the frames with a much cheaper interpolation is much easier on the hardware and does not affect the player's gameplay experience at all.

Keep the number of Draw calls at the minimum

Each *Draw()* call on a *SpriteBatch* adds load to the GPU. Implement the engine in a way that it operates with as small amount of *Draw()* calls as possible.

Optional, dedicated, decoupled components

The components of the engine should be responsible for only one thing and do that as efficiently as possible. It must be obvious for a game developer what a component does for and how to use it. Also, components must be decoupled, the game developer should be able to add and remove individual components dynamically as needed, and there should be no dependency between them.

Generalized code, abstract classes, interfaces, generic classes

The code should be written in a generic way to support the widest variety of usages. Using interfaces, abstract and generic classes is a great way to maximize reusability and hide the internal structure of the engine, while also giving an opportunity to redefine certain behaviors of the superclasses when it makes sense.

Hide the unnecessary and internal functionalities

Interfaces, abstract classes, and inheritance is a key technique to define what the game developer should have access to. In general, only those classes, functions and variables should be visible for the game developer which can somehow be relevant for the game development process: provides functionality, configures/alters the behavior, or provides an opportunity for an override. Classes, variables, and functions intended strictly for internal use should be hidden with the proper access modifier (private, internal, sealed). This avoids confusion and prevents misuse. The 'internal' visibility in C# is a great tool to create a variable visible everywhere from the same binary, but not externally (acts as 'public' in the engine source code and as 'private' in the video game source code)

Prefer performance over other human-centered code metrics

Although the human aspect of code quality is always important, for programs like game engines, where performance is critical, especially in the most calculation-heavy components, performance must be preferred over other general code metrics. For example, if a frequently called operation can be done much faster using bitwise operations than regular +,-,*,/, etc. operators, prefer the binary ones, even if it makes the code harder to understand for other people. This goes against usual application development standards, where, if it's not a performance-critical application, simplicity and human readability is often more important than writing the given piece of code as efficiently as possible, if that would result in a code that is very hard to understand and therefore hard to maintain and support.

1.4 Achievements

Apart from the particle system and object pooling, at the time of writing this thesis, all the above-mentioned components got implemented. The engine is functional, a platformer game demo demonstrates the capabilities and usability. Although the game itself has been tested on PC and Android platforms, only the PC version is publicly available as of now. Below I will describe the most important components and solutions of the engine.

2. Implementation of the game engine

2.1 The game loop

Implementation: Engine/Source/Game/MonolithGame.cs

The main loop of the engine is driven by MonoGame: *Update(GameTime gameTime)* and *Draw(GameTime gameTime)* calls follow each other repeatedly as long as the game is running. They are always next to each other: there are never two consecutive *Update()* or *Draw()* calls. The *gameTime* input parameter stores how long the game has been running and how long did the previous frame took to render, both can be obtained in different time units from the parameter (millisecond, second, etc). The frequency of these calls is driven by the capabilities of the hardware: the more frames the hardware can produce, the more calls will be made in each second. Typically, mobile phones and video game consoles run video games with a fixed 30 or 60 frames/second upper cap, while a modern PC can produce even thousands of frames/second.

This leads us to our next problem: providing the same experience with variable frame rates.
Let's say we want to move an object on the screen with the following code:

```
/*
Vector2 is a struct in MonoGame, where the (3,5) means 3 units on the X-axes and 5
units on the Y axes. The addition, subtraction, multiplication, and division
operators are overloaded to work between Vector2-Vector2 and Vector2-decimal data
types.
*/

private Vector2 acceleration = new Vector2(0, 1);
private Vector2 velocity = Vector2.Zero;
public void Update(GameTime gameTime) {
	velocity += acceleration;
	object.Position += velocity;
}
```

MonoGame uses a left-to-right, top-to-bottom coordinate system: the top left corner of the screen is at position (0,0), and the X coordinate increases by moving right on the screen, the Y coordinate increases by moving down on the screen.

If a game is running with 30 FPS, the object on the screen will move 30 units/second, but on another system, if the game is running with 120 FPS, the object will move 120 units/second, which is an entirely different behavior and makes the game unplayable on any other frame rate than it was designed to.

Solution: Semi-implicit Euler method

Source: https://gafferongames.com/post/fix_your_timestep/

In many video games and engines, this is the preferred solution for the above-mentioned variable framerate problem.

What is velocity? The rate of position change over time:

$$dx/dt = v$$

This means that if we know the position and velocity of an object, we can calculate the integral to find its position at some point in the future.

The semi-implicit Euler method can help us to solve a system of ordinary linear equations, but instead of calculating the integral, we are going to numerically solve it. The above equation can be rewritten like this, let *dt* be delta time:

```
velocity += acceleration * dt;
position += velocity * dt;
```

Leading us to the following implementation of the movement:

```
private Vector2 acceleration = new Vector2(0, 1);
private Vector2 velocity = Vector2.Zero;
public void Update(GameTime gameTime) {
	// let dt be the total time it took to render the previous frame
	float dt = gameTime.ElapsedGameTime.Milliseconds;
	velocity += acceleration * dt;
	object.Position += velocity * dt;
}
```

If a game runs with 30 FPS on a system, the value of *dt* will be 33.3 (it takes 33.3 milliseconds to render one frame). The movement in 1 second:

30 (framerate) * 33.3 * 1 = 999.
This means that the game object will move 999 units in 1 second with 30 FPS.

If the game runs with 60 FPS on a different system, *dt* will be 16.6 and the movement in 1 second:
60 (framerate) * 16.6 * 1 = 996.

Although, because of the rounding, these values are not exactly the same, it's close enough for most video games. By changing the value of *acceleration*, we can change how fast the object moves on the screen and it will look the same on all systems, regardless of the framerate.

This fixes the movement of an object with variable framerate but does not solve all the problems we can encounter with variable frame rates. Some physics simulations can behave differently due to different delta-time values, from having a slightly different feel of the game to a spring exploding to infinity.

Solution 1: limit the FPS.
Many games choose the approach of having an upper bound of how many frames the engine will render in each second. If we cap the framerate to 30 or 60 FPS, there will be no big delta-time differences between different systems and the game will behave predictably and the differences in the physics will be subtle to non-existing.
Although this approach works, it limits the player's gameplay experience. Today's computers can generate thousands of frames/second and high-end gaming monitors can operate up to 240 Hz, meaning that they can display 240 frames/second. Even a person that does not play video games can easily tell the difference between 30 FPS and 60 FPS gameplay just by the fluidity of the rendering, but an experienced player will notice frame rate differences on even higher values, and if we limit the framerate of the game, they won't be able to utilize their hardware properly and we deprive the game itself of its full potential, therefore, we are going to choose a different solution.

Solution 2: fixed timesteps

Instead of capping the rendering of the game, let's introduce a new type of update method that will be called with a fixed frequency of 30 times/second, regardless of how fast hardware can render and do game logic updates in it. I will call it *FixedUpdate()* and will refer to it like that for the rest of the thesis.

Introducing the *FixedUpdate()* will benefit us in the following ways:

- Physics simulations will be predictable and will behave exactly the same way on each hardware, regardless of how many frames can be rendered on each system.
- Running a game can require a series of complex, hardware-heavy calculations which, in the normal *Update()* loop, would be calculated as many times as the frames rendered, which is not just unnecessary, but also puts a load on the CPU.

  For example, there is no point in running hundreds, or maybe thousands of collision checks, physics updates, etc. for an average game. Let's take this one step further: there is no point in running most gameplay related logic more than 30 times/second, as the users won't notice it, but it would put a heavy load on the CPU, which is especially a problem for mobile phones and handheld gaming consoles: it drains the battery very fast and can heat up the device.

  In this engine, we will compute the CPU-heavy computations only 30 times/second (configurable) and do an interpolation for the in-between frames, so players will still experience smooth gameplay while going easy on the CPU.

The *FixedUpdate()* loop will be implemented the following way:

We are going to measure the elapsed time of the game in an *accumulator* variable in the normal *Update()* method. Whenever the value of this variable is higher than our configured *FixedUpdate()* frequency, we are going to run one *FixedUpdate()* call and decrease the value of the accumulator by the frequency. This guarantees that the *FixedUpdate()* will be called whenever it's due, maybe even multiple times to compensate for the accumulated time fractures resulting from doing a discrete event in continuous time. We are also going to maintain a value called *ALPHA*, which will be used for linear interpolation. *ALPHA* will have a range from 0 to 1 to measure how far we are until the next *FixedUpdate()* call. For example, *ALPHA* value 0.2

means we are 20% in the current call, we had a *FixedUpdate()* recently, and *ALPHA* value 0.5 means we are exactly halfway between the last and next upcoming *FixedUpdate()* calls.
From now on, we will move our object in the *FixedUpdate()* call instead of the regular *Update()* call.

```
// this will be the FixedUpdate() frequency, we set it to 30 FPS
private float fixedUpdateDelta = (int)(1000 / (float)30);

// helper variables for the fixed update
private float previousT = 0;
private float accumulator = 0.0f;
private float maxFrameTime = 250;

private float ALPHA = 0;

override void Update(GameTime gameTime)
{
    // only relevant in the very first call
    if (previousT == 0)
    {
        previousT = (float)gameTime.TotalGameTime.TotalMilliseconds;
    }

    float now = (float)gameTime.TotalGameTime.TotalMilliseconds;
    float frameTime = now - previousT;
    if (frameTime > maxFrameTime)
    {
        frameTime = maxFrameTime;
    }

    previousT = now;

    accumulator += frameTime;

    while (accumulator >= fixedUpdateDelta)
    {
        FixedUpdate();
        accumulator -= fixedUpdateDelta;
    }

    ALPHA = (accumulator / fixedUpdateDelta);
    base.Update(gameTime);
}

public static float fixed_dt = 1;
public void FixedUpdate() {
	velocity += acceleration * fixed_dt ;
	object.Position += velocity * fixed_dt;
}
```

And the *Draw()* method would look like this:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.White);

    spriteBatch.Begin();
    spriteBatch.Draw(object.Texture, object.Position, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

Notice that we have set the *fixed_dt* to 1. In the *FixedUpdate()*, *elapsedTime* is meaningless as the frame times are always the same. But if we multiply with *fixed_dt* everywhere in the code where we would normally multiply with the real *elapsedTime*, we can later change the value of fixed_dt to a lower value to create a very cool bullet-time effect! Bullet-time is an effect that became widely known from the movie Matrix: it's the effect where the time slows down. If we change the fixed_dt to 0.5, everything will move/happen 50% slower in the game.

Let's inspect how our program behaves now. Assume that the *FixedUpdate()* is configured to 30 FPS and the game itself is rendered at 120 FPS. This how the event flow would look like:
FixedUpdate(): updating the position of the object to (100, 100)
Draw(): drawing the object at position (100, 100)
Draw(): drawing the object at position (100, 100)
Draw(): drawing the object at position (100, 100)
Draw(): drawing the object at position (100, 100)
FixedUpdate(): updating the position of the object to (100, 120)
Draw(): drawing the object at position (100, 120)
Draw*()*: drawing the object at position (100, 120)
Draw(): drawing the object at position (100, 120)

Draw(): drawing the object at position (100, 120)

FixedUpdate(): updating the position of the object to (100, 140)

Draw(): drawing the object at position (100, 140)

And so on…

How would this look on the screen? Even though the computer renders 120 frames each second, we only update the position of the object 30 times/second. This means that the object will be drawn in the same position for 4 consecutive frames until *FixedUpdate()* is triggered and updates the position again. So the player will see that although the computer is rendering with 120 FPS, the gameplay feels like 30 FPS.

Solution: linear interpolation.

Instead of drawing the object to the actual position, let's do a linear interpolation between the previous position and the current position using the *ALPHA* value described earlier.

Let's modify the *FixedUpdate()* method to save the previous position of the object:

```
public static float fixed_dt = 1;
public void FixedUpdate() {
	object.PreviousPos = object.Position;
	velocity += acceleration * fixed_dt ;
	object.Position += velocity * fixed_dt;
}
```

And let's change the *Draw()* method to draw to the interpolated position:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.White);

    Vector2 drawPosition = Vector2.Lerp(object.PrevPosition, object.Position, ALPHA)
    spriteBatch.Begin();
    spriteBatch.Draw(object.Texture, drawPosition, Color.White);
    spriteBatch.End();

    base.Draw(gameTime);
}
```

This will result in smooth rendering, as, instead of rendering the object onto the same position 4 times, we will interpolate its position according to *ALPHA* value and draw it onto this interpolated position, which is different in every frame.

The player is experiencing smooth gameplay while the update is happening only 30 times/second.

Why do we do this interpolation instead of just simply using the *Update()* method to update the object's position more frequently? Because in an actual video game, determining the position is often the result of a series of very expensive calculations: collision detection, physics, object-to-object interactions, AI, and so on, and running these expensive calculations only 30 times/second results in a predictable behavior with as low CPU usage as possible, while, thanks to the interpolation, the user will still experience smooth gameplay not even knowing that the game logic is only updated with 30 times/second. And, as described above, lower CPU usage means lower power consumption, resulting in longer battery life and less heat.

However, there is one challenge to solve when using *FixedUpdate()*: collision detection. Detecting collisions is a discrete calculation, which means that it always happens at the current position of the object. If we are running the game logic in a *FixedUpdate()* configured to 30 calls/second, it also means that we are only doing 30 collision detections/second. While this might seem good enough for some games, many video games have fast-moving objects (like bullets), where the positions change quite a lot from one *FixedUpdate()* to another, and this change in position might be bigger than the size of the collider, meaning that we skip some collisions. This is a problem because if the player is shooting at a small enemy, the collision must be detected.

Solution 1: Increase the *FixedUpdate()* frequency

While this might work for certain games, where the colliders are relatively big and the objects' movement speed is capped, in general, this is not a good solution. Even if we increase the *FixedUpdate()* from 30 to 60 calls/second, thus resulting in twice as much collision detection, we did not solve the problem, just pushed it away so we encounter it in a later situation where there is an even faster-moving object or a smaller collider. Also, by increasing the *FixedUpdate()*

frequency, we are putting extra load on the CPU which, because of the reasons described earlier, is a bad thing.

Solution 2: run a collision detection for the interpolated, in-between frames too

While this would improve the situation a lot, and would probably be enough for most games, this would mean that if a game is rendered with 120 FPS, we would do 120 collision detections/second, which would defeat the purpose of having a fixed timestep, we might as well just delete the whole *FixedUpdate()* logic and run everything in the normal *Update()* method, so this also not a good solution. Also, it still doesn't guarantee that all collisions will be detected.

Solution 3: step-based movement logic

Implementation: Engine/Source/Entities/PhysicalEntity.cs

The real solution is step-based movement. For this, we will need to decide what is the smallest collider that we guarantee to get detected, let's pick 16x16 pixels as this seems to be enough for the vast majority of video games. Although at first thought, the developers might think "I want to detect a collision even if it's just 1 pixel!", hardly any game needs collisions this small. In real life, colliders have a bigger size, 16x16 is the usual minimum which is easy to calculate but accurate enough to provide a great gameplay experience.

The solution is the following: we will guarantee that it doesn't matter how fast an object is moving, we will always detect collisions that are at least 16x16 pixels. We are going to achieve this by introducing a stepper logic: whenever we move an object on the screen, we will move at most 16 pixels in one step, guaranteeing that collisions are always detected. If the velocity is higher than this, we will move the object in several, smaller steps and do collision detection in each step.

Examples speeds:

- The X speed of the object is 12: we will move the object 12 pixels to the right and do a collision check.
- The X speed of the object is 16: we will move the object 16 pixels to the right and do a collision check.

- The X speed of the object is 18: we will move the object 9 pixels 2 times to the right and do a collision check in both steps.
- The X speed of the object is 36: we will move the object 12 pixels to the right 3 times and do a collision in all 3 steps.

This will guarantee that we will never move more than 16 pixels in one step, which is the minimum size of our colliders.
Also, we did not need to universally increase the *FixedUpdate()* frequency: the extra collision detection will happen only for the objects that are moving fast, and only while they are moving above the threshold. The rest of the game objects are completely unaffected and our *FixedUpdate()* still ticks with 30 calls/second.

2.2 Rendering

Rendering happens through MonoGame's *SpriteBatch* objects: a *SpriteBatch* is a group of 2D images that are drawn with the same setting by the GPU. Since opening and closing a sprite batch has a notable overhead, it is ideal to keep the number of sprite batches as low as possible during gameplay, as well as the number of *Draw()* calls on the sprite batches. Although in theory, it is possible to draw everything via one sprite batch, technically it's rarely double as different sprites have different settings that we want to apply when drawing. But a general rule of thumb is the fewer sprite batches, the better.
Usage of sprite batches:

```
spriteBatch.Begin(...) // opening a new batch for rendering
spriteBatch.Draw(...) // adding sprites to the batch
spriteBatch.Draw(...) // adding sprites to the batch
spriteBatch.Draw(...) // adding sprites to the batch
…
spriteBatch.End();
```

The *Draw()* call of sprite batches can be parameterized in many ways, let's see the most important parameters of them:

- *Texture2D* texture: a *Texture2D* object that will be drawn on the screen. A *Texture2D* is based on a *Color* array, *Color* is a struct that contains 4 channels ranging from 0 to 255: Red, Green, Blue, and Alpha channels.
- *Vector2* position: a *Vector2* that describes where the object will be drawn
- *float* rotation: the rotation of the sprite
- *Vector2* origin: the origin of the rotation
- *float* scale: the scale of the object
- *SpriteEffects* spriteEffects: different effects that can be applied for the sprite, like horizontal or vertical flipping
- *Rectangle* sourceRectangle: when there is no need to draw the whole texture, we can specify a rectangle that will tell the sprite batch which part of the texture we want to draw.

A *SpriteBatch* can also be opened with many parameters, but I will only mention one of them now: *SpriteSortMode*. This configures how the sprites are sorted within the same batch. We can configure it to sort sprites based on the *Depth* parameter to set a draw order, but I'm using *SpriteSortMode.Deferred* everywhere in the code. This means that MonoGame will not do any kind of sorting when rendering, it will draw the sprites in the order of the *Draw()* call sequence (without sorting). Using anything other than *SpriteSortMode.Deferred* has a performance impact, as MonoGame would sort all the textures in every single *Draw()* call based on the *Depth* variable, which, in case of many textures, is an expensive operation. I have implemented my own way of sorting sprites, which is much more effective than MonoGame's built-in sorting modes, therefore I'm using *SpriteSortMode.Deferred* for each of my sprite batches. The sorting method will be described later in the *Layers* section.

As I mentioned above, *SpriteSortMode.Deferred* means that the sprites will be drawn based on the *Draw()* call sequence. Let's take an example, 3 objects are drawn in the same position:

```
spriteBatch.Draw(object1.Texture, position, Color.White);
spriteBatch.Draw(object3.Texture, position, Color.White);
spriteBatch.Draw(object2.Texture, position, Color.White);
```

In this scenario, *object2* will be drawn on top, *object3* would be drawn behind *object2* and *object1* will be behind *object3*, meaning that *object2* will partially or fully cover *object3* and *object1*.

In the following, I will present the most important classes and components of the engine.

2.3 Camera

Implementation: Engine/Source/Camera/Camera.cs

The camera class provides basic functionalities that most games need: scrolling, zooming, following an entity with the given parameters (to achieve smooth, natural camera movement), and shake (in case of explosion and heavy objects landing). The class is more of an abstraction than a real camera: it does not actually do rendering, but it provides a transformation matrix for the renderer. In MonoGame, *SpriteBatches* can be rendered using transformation matrices, which will drive how the current frame is displayed. The camera has a *position*, a *zoom* level, a limit *Rectangle* (limits the camera scrolling between certain values, so it won't show anything outside of the intended gameplay area), a *Viewport* (we don't necessarily want to render in the whole window, we might want to add multiple cameras for local multiplayer games, and all players would have their own *Viewport* on the screen) and using these, the camera calculates the transformation matrix that is passed to the sprite batches in each frame, also taking into consideration the different scroll speeds of the parallax backgrounds (parallax background are background layers that scroll slower than normal layers, creating a sense of depth in the background). Although the camera has a position that is updated when it's tracking an entity, it never moves in the world space, it's only needed to create the transformation matrix for the rendering. Passing this matrix into the sprite batches will result in a properly working game with scrollable backgrounds and adjustable zoom level.
The camera also provides a secondary, static transformation matrix for UI elements: this is only updated whenever the screen resolution is changed, otherwise, it's static because UI elements are on a fixed position on the screen and their size is also not affected by the *zoom* level.

2.4 Layers

Implementation: Engine/Source/Layer/Layer.cs and LayerManager.cs

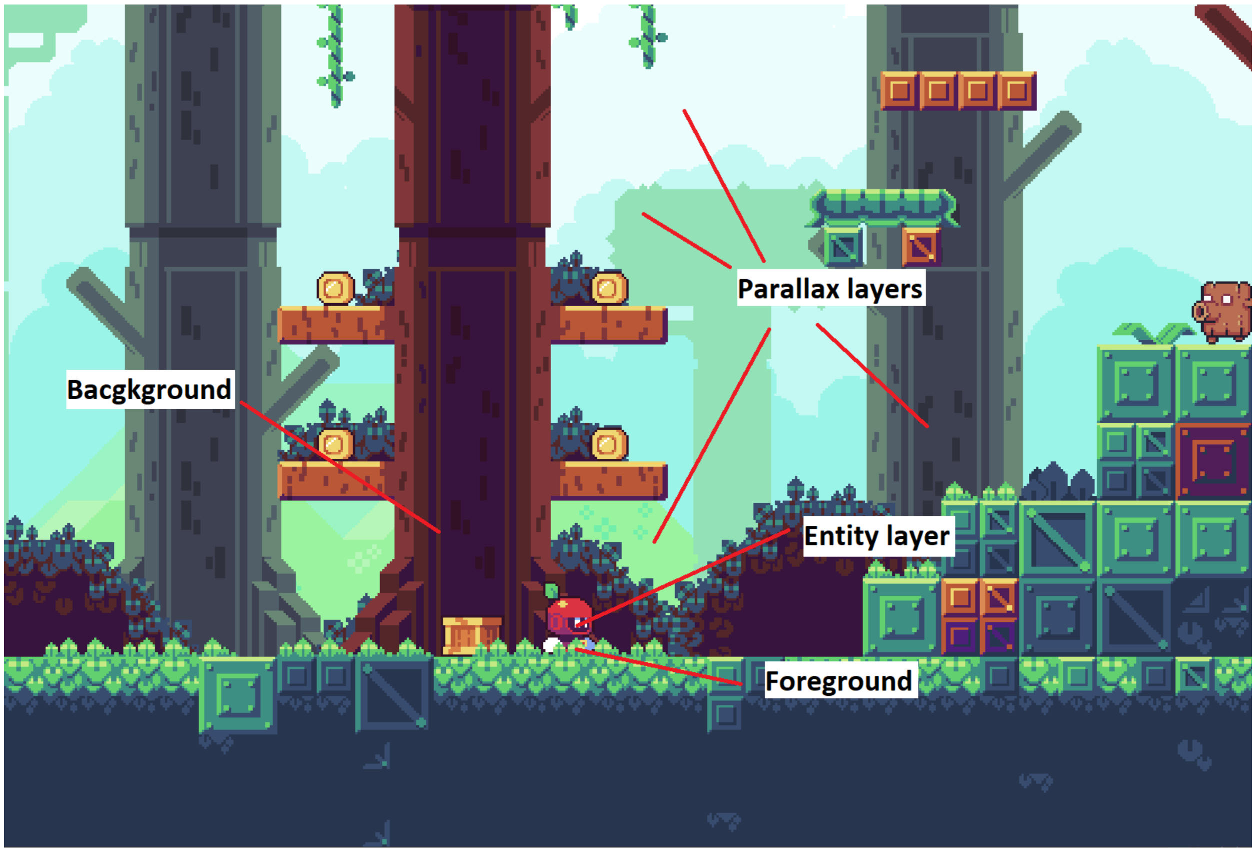
Layers are groups of entities that must be drawn with the same settings. Each layer has its own *SpriteBatch*, and each layer opens and closes the sprite batch and calls *Draw()*, *Update()*, and *FixedUpdate()* method on the entities assigned to it. Each layer must be handled by and assigned to the *LayerManager*. Whenever a new layer is created, the *LayerManager* sorts the layers to guarantee proper draw order and whenever an object's *drawPriority* is changed, the layer sorts its list objects to guarantee proper draw order among its entities. This is much better than letting MonoGame do the ordering: MonoGame would do it for each frame and every entity, while the engine only does it when a new layer is created or an object's *drawPriority* is changed, and both of them are typically happening during startup. Ordering during gameplay almost never happens, except when Y-sorting is turned on for the layer, but in that case, it's inevitable. Y-sorting is a method to draw entities based on the Y position. It can be useful for top-down games where, if the hero is below a tree (Y coordinate is bigger than the tree's Y coordinate), the hero should be drawn on top and partially cover the tree, but when the hero is higher than three (Y coordinate is smaller than the tree's Y coordinate), the tree should be drawn on top and partially or fully cover the hero. This gives a sense of depth. There is one optimization as well here: we only do Y-sorting if any dynamic entity's Y-position is updated.

Visual representation of Y-sorting

Layers are rendered in a strict order:

1. parallax background layer(s) (clouds, mountains, distant trees, etc.)
2. background layer (ground, obstacles, vegetation, etc.)
3. entity layer (player character, enemies, traps, etc.)
4. foreground layer (vegetation, environment, etc.)

As the rendering is happening in the above order, the foreground layer will be fully visible on the screen, then the entity layer, then the background layers, and lastly the parallax layers. With the layers, we can achieve a sense of depth: if we add a rock or a bush to the foreground layer, the entities will be hidden by them (the hero can "hide" behind them for example), while entities cover parts of the background layer, and the background layer covers parts of the parallax layers. When the *Pausable* flag is set to true, layers can also be *Paused*, which means that the *Update()* and *FixedUpdate()* methods are not going to be called on them. This is very useful when displaying texts on the screen waiting for user input to continue: while the user is reading text, the game is paused and the user can press a button to carry on playing.



Visual representation of the layers

2.5 Scenes

Implementation: Engine/Source/Scene/AbstractScene.cs and SceneManager.cs

The game can be divided into separate scenes that contain layers, UI, camera, and other components that are necessary to run the game. The menus can be separate scenes, as well as maps and areas of the game. In the example platformer game, the menus have their own scenes, and both level 1 and level 2 are separate scenes. Each scene calls *Update()*, *FixedUpdate()*, and *Draw()* on its layers, and updates and draws the UI as well.
Scenes can be created very easily with the engine: they must extend the *AbstrctScene* class and there are methods to override to load assets and initialize game logic, and events to trigger when a scene ends.
A *SceneManager* is responsible for handling the scenes, therefore each scene must be added to the manager. Scenes not added to the manager are considered non-existent (and will be garbage collected). The *SceneManager* handles scene transitions automatically, it loads and unloads scenes whenever it's necessary, so the game developer never has to care about manually cleaning up after a scene has finished: the engine will destroy all object references and the garbage collector will free up the memory. There is always one "active" scene that is being rendered called *CurrentScene*, but multiple scenes can be updated next to the *CurrentScene*. This is especially useful when making a top-down game: the character walks into a town where the outside areas are one scene and the inside areas (taverns, houses, etc.) are separate scenes. While the character is inside a tavern, for example, we still want the rest of the town to be updated: the NPCs should carry on with their daily routines in the outside areas, the time of the day should also pass, not just inside the tavern, so when the player walks out of the tavern, it won't feel like the world was paused while the character was inside.
Scenes can be configured for the most typical scenarios:

- To get updated while it's not the current scene (like the previous top-down game example)
- To use a loading screen: while the assets are being loaded, a loading screen will appear instead of "freezing" the screen

- To preload all the assets for the scene at the game's startup: some games benefit from having all the scenes preloaded when starting the game. Like the above-mentioned top-down game example, players rather wait a bit more at the game's startup rather than seeing loading screens every time they go into or out of the tavern. While, on the platformer game example, one scene is one level, there is no need to preload them, the player will be okay to see a short loading screen when they complete a level and go to the next one.

Scenes can be started in 2 ways:

- *LoadScene*: loads all the assets, starts the scene, and unloads (destroys) the current scene. In the platformer game, when going from level 1 to level 2, this method is called as we will not go back to level 1 from level 2, so we can unload level 1 before moving to level 2.
- *StartScene*: when a scene is already loaded, and we just want another scene to be the active scene, without unloading the current scene, this is the method to call. In the top-down game example, going into and out of the tavern should use the *StartScene* for the transition, as we need all the scenes to stay in memory and we want to be able to switch between them without having to reload all the assets and losing their state.

2.5 Entity and PhysicalEntity

Implementation: Engine/Source/Entites/Entity.cs and PhysicalEntity.cs

The *Entity* class represents a game entity that can be updated and/or rendered by the engine. It has two important bool flags: *Active* and *Visible*. If *Active* is true, the *Update()* and *FixedUpdate()* functions of the entity will be called, if *Visible* is true, the *Draw()* function will also be called. If the game developer does not want the entity to be visible, they can just set the *Visible* flag to false, if it's only visible and there is no logic attached to it, the *Active* flag can be turned off.
Entities have a *Transform* class member, which contains the *Position* of the object, but it does not have velocity! This means that this class should only be used for static objects which never

need to change their position in the world space. (like traps, pickups, etc.). Different kinds of *Components* can be attached to an entity, they will be explained later. Each entity calls *Update()*, *FixedUpdate()*, and *Draw()* on the components attached to it (whichever is applicable) whenever the *Active* and/or *Visible* flags are true. The game developer can extend this class to create a game object that they would like to display and/or update in the game. The child classes can override *Update()*, *FixedUpdate()*, and *Draw()* methods to add their own logic.

PhysicalEntity extends from *Entity*, the main difference between them is in the *PhysicalEntity* class, the *Transform* member also contains *Velocity*, and changing this will change the object's position. While this seems like a very minor difference between the 2 classes, movable objects mean quite a lot of extra game logic to implement (forces, collisions, gravity), and this class contains these. Also, if there is a collision component attached to the *PhysicalEntity*, collision checks will be performed during each *FixedUpdate()* loop for these objects.

Physical entities can be moved by adding to/removing from the *Transform.Velocity*, which is also corrected by horizontal and vertical friction values to provide a more natural feel of movement for the player. When turned on, gravity is also applied to the *Velocity*. By adjusting the gravity and the friction, it's possible to finetune an object to give a different sense of mass: a bird's feather will fall differently than a metal ball, and also, moving on ice is more slippery than moving on solid ground. Both can be expressed by adjusting the friction and gravity values. Later, this will be refactored by introducing a "mass" variable to finetune the object's behavior.

Entities and physical entities can also be in a parent-child relationship with each other: the child will inherit the parent's *Position* and their own *Position* and *Velocity* will be relative to the parent's. Also, it's guaranteed that the parent will always be updated and rendered first, then the children. Each object can have one parent and unlimited children.

Extending from these classes is an easy way to create objects in the game that can be moved around and used for various gameplay purposes.

Also, both *Entity* and *PhysicalEntity* instances can have tags. Tags are used for various purposes, for example, for collision optimization. Let's say the player has a main character that they want to collide against enemies. In this case, the player character class has to call `AddCollisionAgainst("Enemy")` in the code and all the enemy classes should add `AddTag("Enemy")` in their own code (having an abstract enemy class doing this and other common things is a good technique). This way, whenever a collision event is triggered between the hero and the enemies, the respective callbacks will be called, but collision checks won't be performed unnecessarily for objects that no one is interested in.

2.6 Components

Implementation: Engine/Source/Components/IComponent.cs and ComponentList.cs

Components are objects that have a certain responsibility/functionality and can be attached to/detached from all types of entities, independently from each other. They all share a common interface called *IComponent*, and they can also implement *IUpdateableComponent* or *IDrawableComponent*, which marks if the component needs to be included in the owner entity's *Update()*/*FixedUpdate()* and/or *Draw()* loops.
Currently implemented components, all will be detailed later:

- *Sprite*: represents a 2D image that will be rendered
- *AnimationStateMachine*: a state machine that plays animations based on the owner's state (also rendered)
- *BoxCollisionComponent*/*CircleCollisionComponent*: collision components for dynamic collision detection
- *BoxTrigger*: a trigger component that fires whenever an entity's position is inside the trigger.
- *AIStateMachine*: a state machine that can be used to efficiently implement AI

These can be added and removed as needed for each entity, and where it's applicable (sprites, animations, box collision components, and triggers) moves together with the entity as well. For example, if an entity needs a static image, a *Sprite* component can be attached to it and the engine will render it and if it also needs circular collision detection, a *CircleCollisionComponent* can also be attached (and removed if not needed anymore).
If a *CollisionComponent* is attached to the entity and the tags are added properly, by simply overriding the *OnCollisionEnter(...)* and *OnCollisionEnd(...)* methods of the superclass, the object will be notified whenever a collision event occurred/ended with the necessary details.
Similarly, when a trigger component is attached, by overriding *OnEnterTrigger(...)* and

OnLeaveTrigger(...) methods of the superclass, the object will be notified when a trigger has fired/ended with the necessary details.
This ensures that the developer has fine control over what is happening with the object, but it's not overburdening and very simple to use and understand.

2.7 Collisions

Implementation:

Single-pont collision: Engine/Source/Entities/PhysicalEntity.cs

Dynamic collisions: Engine/Source/Physics/CollisionEngine.cs and Engine/Source/Physics/Collision/*.cs

There are 2 parallel collision detection methods running in the engine: a single-point collision system and a dynamic collision system.
The single-point collision system is a very cheap, but very effective way to check for collisions against a dynamic entity and a static entity, like the player character and the static parts of the environment. The idea is the following: the gameplay area is split up to a 2D grid and each object has a GridPosition maintained, which is an integer and returns in which grid cell is the object currently located. For dynamic objects (that are extended from *PhysicalEntity*), this grid position is updated with the movement update logic. The check is simple: check the left, right, top, and bottom grids around the dynamic character whether it contains a collider, and if it does, we zero out the velocity in that direction. This is very fast to compute and accurate enough for the majority of video games for environmental collision.
Dynamic collisions are different, it's a collision calculation between two dynamic objects. Collision components must be attached to the entities to get detected, and the *AddCollisionAgainst(string tag)* method will trigger a collision check for the given object. Respectively, we have to add the same tag to the objects that we want to detect. If the object is not colliding against any tags (*AddCollisionAgainst* never called) it means that there is no need to do collision checks for this object, but it can still be detected by other objects if it has a tag that someone collides against. This is an optimization technique to do as few collision checks as possible against objects, as it's a very expensive operation and we want to avoid doing it unnecessarily.

Currently, 2 types of collision components can be added to an entity: rectangle and circle, the shape of most objects can be efficiently approximated with these 2 primitives, but more shapes will be added in the future (like a triangle) for more accurate collisions. The collision between the shapes is calculated based on their position, and it's an accurate collision, meaning that whenever the shapes touch/intersect, a collision callback will be triggered.
Spatial hashing will also be implemented later: another level of collision detection optimization: collisions will only be checked against objects that are relatively close to each other.

2.8 Triggers

Implementation: Engine/Source/Physics/Trigger/AbstractTrigger.cs and BoxTrigger.cs

Triggers are similar to dynamic collisions, but it's simpler than that: it's a shape against a point. Currently, only rectangular trigger is implemented and usually, it's enough for most games, but for special needs, circle shape will also be added later. The collision check is simple: we check whether the other object's position is inside the bounds of the shape and do the respective callbacks. Since it's based on the position, which is a point, it does not check against the visible shape of the objects, but it's very cheap to calculate and good enough for many scenarios (like walking into a room or an area that opens a door or being in a certain vicinity of an enemy).

2.9 Sprite

Implementation: Engine/Source/Graphics/Sprite.cs

Sprites represent 2D images that, when attached to an entity, can be rendered. It contains the *Texture2D* object that is drawn, uses the position of the owner entity (plus an optional offset), rotation, sprite effects, source rectangle, and some other useful parameters to customize rendering.
Sprites are written in a way that when the input image is provided, it will automatically process them to find the smallest source rectangle that is necessary to render it which can also be used as a default box collision shape. It's also possible to generate the circumscribed circle for circle collision components.

2.10 Animations

Implementation: Engine/Source/Graphics/Animations/*.cs

SpriteSheetAnimation

2D animations are usually stored on one image called sprite sheet: each frame of the animation is put into the same image. The format is usually PNG as it supports transparency, which is very useful for video game art. The image is logically split into rectangles: for example, an animation with 15 frames can be 2x8 rows: the first row has an animation frame in each rectangle, the second row has animation frames in 7 rectangles, the last rectangle is empty. The *SpriteSheetAnimation* class can determine the size of the rectangles if they are all powers of 2, but if not, this rectangle size can also be provided manually. Also, the class automatically detects which rectangles have an image in them and automatically constructs the animations from the non-empty frames, the game developer only needs to provide the desired frame rate of the playback. Also, callbacks can be defined for each animation, for example, to trigger an event when the animation starts/ends or when it's at a certain frame. Animations can be parameterized in several ways, like enable/disable looping, pause condition, etc.

Sprite sheet example: the carrot's movement

SpriteGroupAnimation

It's the same as *SpriteSheetAnimation*, the only difference is that the input source of the animation frames is not one sprite sheet, but multiple separate images.

AnimationStateMachine

The animations are driven by the *AnimationStateMachine* class. Each animation has to be added to a state machine, together with a unique name, condition, and an optional priority.
The condition is the most important part of this class: we can specify which state the entity has to be in to play the given animation.
Let's take an example, we have 6 animations: running left, running right, jumping left, jumping right, falling left, falling right.
Inside the class' constructor (to handle movement, the class must be inherited from *PhysicalEntity*), we have to add the following code:

```
AnimationStateMachine anim = new AnimationStateMachine();
AddComponent(anim);

SpriteSheetAnimation runningLeftAnimation =
      new SpriteSheetAnimation(Assets.GetTexture("CarrotRunLeft"), 24);

SpriteSheetAnimation runningRightAnimation =
      runningLeftAnimation.CopyFlipped();

//… assuming we have prepared the rest of the animations…
SpriteSheetAnimation jumpLeftAnim = …
SpriteSheetAnimation jumpRightAnim = …
SpriteSheetAnimation fallingLeftAnim = …
SpriteSheetAnimation fallingRightAnim = …

// let's add the left movement
bool isMovingLeft() => IsOnGround && Velocity.X != 0 && CurrentFaceDirection ==
Direction.WEST;
anim.RegisterAnimation("RunningLeft", runningLeftAnimation, isMovingLeft);

// right movement with priority
bool isMovingRight() => IsOnGround && Velocity.X != 0 && CurrentFaceDirection ==
Direction.EAST;
anim.RegisterAnimation("RunningRight", runningRightAnimation, isMovingRight);

// left jump
bool isJumpingLeft() =>  !IsOnground && Velocity.Y <= 0 && CurrentFaceDirection ==
Direction.WEST;
anim.RegisterAnimation("JumpingLeft", jumpLeftAnim, isJumpingLeft);

// right jump
```

```
bool isJumpingRight() =>  !IsOnground && Velocity.Y <= 0 && CurrentFaceDirection ==
Direction.EAST;
anim.RegisterAnimation(“JumpingLeft”, jumpRightAnim, isJumpingRight);

// falling facing left
bool isFallingLeft() =>  !IsOnground && Velocity.Y > 0 && CurrentFaceDirection ==
Direction.WEST;
anim.RegisterAnimation(“FallingLeft”, fallingLeftAnim, isFallingLeft);

// falling facing right
bool isFallingRight() =>  !IsOnground && Velocity.Y > 0 && CurrentFaceDirection ==
Direction.EAST;
anim.RegisterAnimation(“FallingRight”, fallingRightAnim, isFallingRight)
```

Using this technique, there is no need to bind animations to keypresses or any other user-related event: the state machine will play the animation if the condition is true. If the condition is true for several animations at once, an optional priority parameter can be added to play the one with the highest priority. The greatness in this is that it doesn’t matter whether an AI controls the character, or the player, or none of them (like an enemy falling off from an edge), it’s always the right animation that will be played. Once they are set up in the constructor, the game developer can forget about them. There is, of course, a way to manually trigger animations (like attack or damage animations), in that case, their condition must simply return false and they will never be picked up automatically.

2.11 AI

Implementation: Engine/Source/AI/AIState.cs and AIStateMachine.cs

AIStateMachine together with the *AIState* class supports writing and running artificial intelligence. Both of these classes expect generic parameters, which is the type of entity the AI controls. Each AI state machine controls one entity and can have several AI states added to it. As the entity switches between AI states, the state machine will be running the respective state. AI states written by the game developer must extend the *AIState<T>* class.

Let’s look at how the carrot’s AI works in the demo platformer game.

Implementation of the carrot enemy:
GameSamples/Platformer/Source/Entities/Enemies/Carrot.cs
Implementation of the Bresenham line:
Engine/Source/Physics/Bresenham/Bresenham.cs
Source of the Bresenham line: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

The carrot has an *AIStateMachine* component added to it with the following AI states:
CarrotPatrolState: patrols on an area until it reaches a collider or an edge, then it turns back
CarrotChaseState: when the carrot notices the player, it starts following it. When the player is above the carrot (on a platform, for example), the carrot will stop and wait for the player to jump down. As long as the carrot sees the player, it will chase, if it doesn't see the player, it switches back to *CarrotPatrolState*.

The carrot has the following components attached:

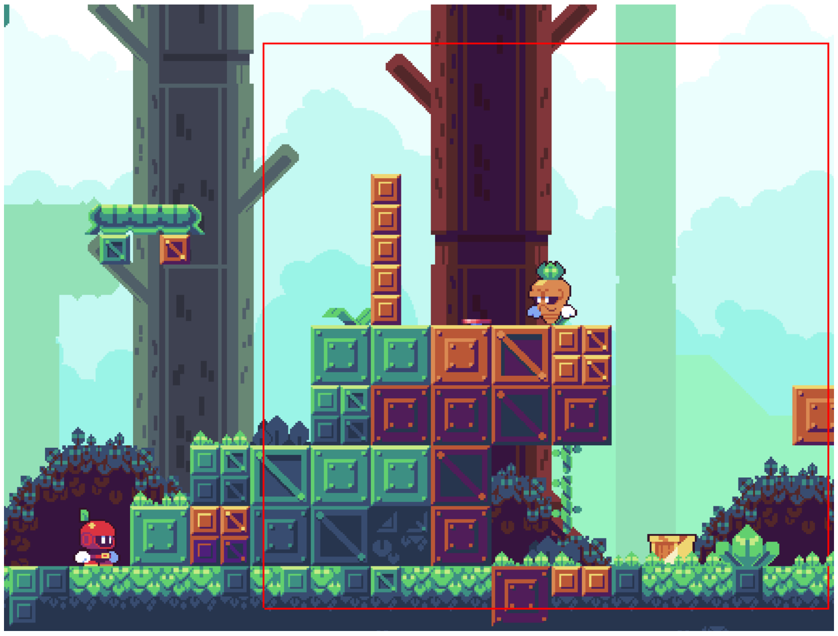
- *AIStateMachine* to run the above mentioned, simple AI
- *AnimationStateMachine* to play the animations
- *BoxCollisionComponent*: to detect whether the carrot is colliding with other entities
- *BoxTrigger*: to detect whether the player is near the carrot.

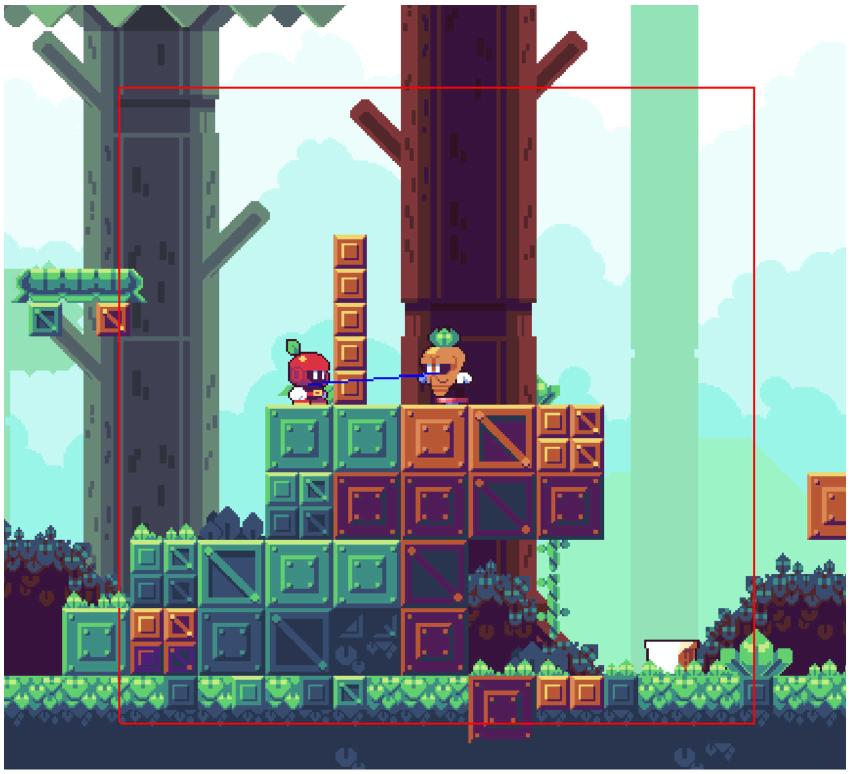
The *BoxCollisionComponent* uses the animation's bounding box to check if the carrot collides with other entities.
The *BoxTrigger* is a much bigger rectangle: it's multiple times the size of the carrot in each direction, and if the player's position is inside this trigger, the carrot will do a raycasting to check whether it actually sees the player or not, and if it sees the player, it will start chasing.
The ray casting happens using a Bresenham line. A Bresenham line algorithm is a very fast algorithm to draw a pixel-perfect line between 2 points. It only works with integers, so it runs very fast and is easy to implement based on the pseudo-codes available online from different sources like Wikipedia.
While the player is inside the carrot's box trigger, the carrot will continuously draw Bresenham lines (always one at a time) from its eyes' position to the player's position. While drawing the line, for each newly calculated point of the line, we check whether there is an environmental

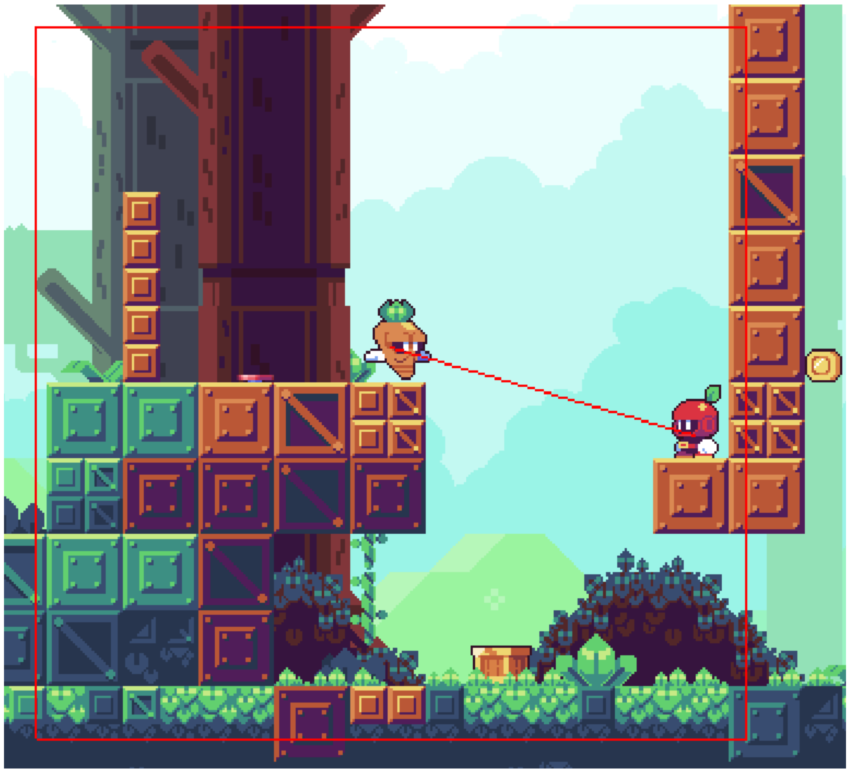
collider on the grid on the point’s position. If there is a collider, it means that although the player is close to the Carrot, it is behind a wall or some other static environmental collider that blocks the vision and the Carrot cannot see the player. If the line can reach the player without encountering any environmental collider, it means that the carrot can see the player and will start chasing. If the player manages to escape from the box trigger of the carrot, the raycasting will stop and the carrot starts patrolling again.
As well as any other game logic-related code, the collision and trigger checks, and the Bresenham algorithm also runs in the *FixedUpdate()* loop, so we will not calculate it more than 30 lines/second for each carrot. This is frequent enough for fluid AI behavior, but still very easy on the CPU.

The Bresenham line can also be used for several purposes in game development:

- Drawing primitives (lines, polygons, even circle)
- AI line of sight (like the carrot)
- Calculating illuminations of light sources: at the game’s or scene’s startup, draw Bresenham lines in 360 degrees and decrease the pixel intensity as we are getting further away from the center, cache result for static objects, and calculate it dynamically only for the most important dynamic objects
- Laser beams: as opposed to regular bullets, laser beams are usually considered immediate and do not need collision checks over time to see if it hits the enemy. When a laser beam is emitted from the player, create a Bresenham line to check whether the points of the line would reach an enemy. This happens in one *FixedUpdate()* call, so time is not a factor here, nothing will move on the screen while the line is being calculated. The line itself could also be rendered to display it like a laser beam, but the visual representation can happen in many ways.


The player is outside of the carrot’s trigger (the big red box), no rays are casted


The player is inside the carrot’s trigger, rays are casted with the Bresenham line algorithm, but the carrot can’t see the player as there is a wall between the player and the carrot (blue line)


The player is inside the carrot's trigger, rays are casted with the Bresenham line algorithm, and the carrot can see the player (red line)

2.12 External map editor integration

Implementation of the map parsing: Engine/Source/Level/Map/LDTKMap.cs

Implementation of the TileGroup: Engine/Source/Graphics/TileGroup.cs

While creating small, simple test scenes is double from source code, creating big, complex maps/worlds are barely possible and it takes away the opportunity from non-programmers (like environment artists and level designers) to productively participate in the video game development process. This is one of the main reasons why most video game engines have a UI, so people can create levels on a 'what you see is what you get basis.

For engines that do not have a UI, it's still possible to design levels/worlds visually by using external map editor software. Many of them are open source and completely free to use for any purpose, so I've decided to natively support my favorite open source level editor called LDtk. Most level editors, including LDtk, works in the following manner:

The user is working on a UI, where they can load all the tile sheets containing the textures. The world space is divided into a 2D grid (usually with a configurable size) and the user can assign textures (or parts of textures) to each position. Then the result is usually stored in some structured text file like XML or JSON. LDtk works based on layers, and (among many other features) it also supports creating entities with properties using the common data types (int, string, boolean, enum, float, etc…). So not only the maps can be edited with it, but also abstract entities can be created and configured, like putting an enemy on the map and setting what items it will drop when destroyed, or how much health it has. The whole map, including all the layers and entities, can be exported to a JSON file, which will describe which part of the texture is on which position of which layer, where the entities are, and what properties they have.
By parsing this JSON, the engine can build the map, place, and configure the entities and layers. For example, if the engine finds an entity with the name “Carrot”, it will create a new instance of the *Carrot* class and configure it with the parameters parsed from the JSON.
I’ve implemented certain naming conventions that, when followed in the editor, will be recognized by the code and processed properly:

- Parallax layers should be named Parallax0, Parallax1… The engine will render them with parallax scrolling.
- Background layers should be named Background0, Background1… The engine will know that it should handle them as background layers and merge them together (this will be described in details later)
- Foreground layers should be named Foreground0, Foreground1… The engine will know that it should handle and render them as foreground layers and merge them together (this will be described in details later)
- The static environmental collision layer must be called Colliders, so the engine will know how to build the collision grid. This layer is not rendered (only for debugging purposes).
- The entity layer must be called Entities, so the engine can return the list of entities for the game developer to handle

The numbers at the end of the layers’ names also define the rendering order.

While the parallax, background, foreground, and collision layers are automatically handled by the engine, the entities can be named as the video game developer wants them, and parsing them will have to happen in the game's code itself.
However, it's not necessary to follow these naming conventions when parsing the map, but the game developer needs to provide their own implementation for the parsing then.

Parsing the map from the JSON:
LDtk outputs a JSON that will (among many other things) will provide the relative path of the tilemap files used and will also tell which position of the tilemap goes to which position of each layer. The source code needs to recognize this and render the background, foreground, and parallax layers.

To understand what the next solution is, we have to know SourceRectangles: *SourceRectangle* defines a certain rectangular part of the texture. By default, it has the size of the whole texture, but we can override it to display just a portion of a texture.
SourceRectange(x, y, width, heigh).
For example, if we have a 32x32 texture, and we want to display the top right 16x16 part of it, we have to pass the following source rectangle for the sprite batch:

```
SourceRectange(16, 0, 16, 16)
```

The performance tests were conducted on a high-end gaming PC in 4K resolution and were always using the exact same scene setup from the same JSON file.

Solution 1
The obvious implementation: for each grid position, let's create a new Entity, create a copy of the texture portion in memory to display in a certain position of the background. As even a small map is typically divided into thousands of grids for each layer, this resulted in thousands of entities and the same amount of small textures and *Draw()* calls created. This had a very bad performance, the test scene produced about 300 FPS.

Solution 2

Let’s not create a new texture for each entity, just pass a different *SourceRectangle* to each of them and all should use the same sprite sheet texture instead of their own small texture portions. This yielded much better results, the test scene produced around 700 FPS, as for MonoGame, switching to render between different *Texture2D* instances has an overhead. That’s why it’s recommended to use sprite sheets, so that internally, most things can be rendered by referencing the same *Texture2D* instance and just pass a different source rectangle. But it was still too slow, we were still creating thousands of entities and so having thousands of *Draw()* calls for each in each loop.

Solution 3

We have to recognize that the Background, Foreground, and Parallax layers are completely static, nothing will ever change position relative to each other within the same layer.

Optimization: let’s merge the tiles in each layer into just one big *Texture2D* instance and just have one entity for each layer.

This is how *TileGroups* were born in the engine: a *TileGroup* is a texture builder tool that can merge texture fragments into one texture. Simply add a *Texture2D* or a *Color[]* into a position, and when there are no more textures to add, calling the *GetTexture()* method will merge these together into one *Texture2D* instance that can be used as a texture, without the need to overwrite the source rectangle.

This has decreased thousands of entities, Texture2D instances, and Draw calls to only 10 (4 parallax layers, 4 background layers, and 2 foreground layers). This has brought tremendous performance increase as it has also decreased the number of *Draw()* calls from thousands to 10: the test scene produced an average of 1700 FPS.

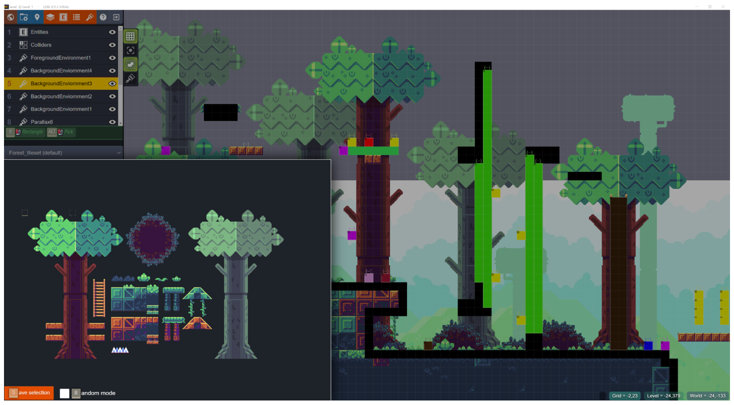
Solution 4

Although the previous solution is already good enough, a game engine can never render too fast, so there was one additional improvement implemented:

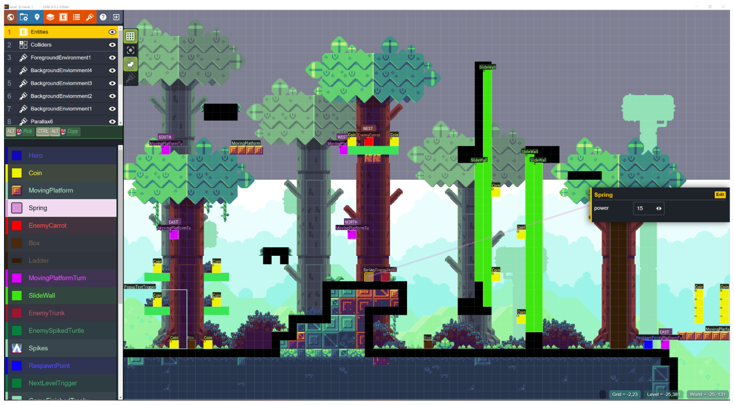
We have to notice that it’s not just that the individual tiles are static within each layer, but also the static layers (background and foreground layers) are completely static to each other!

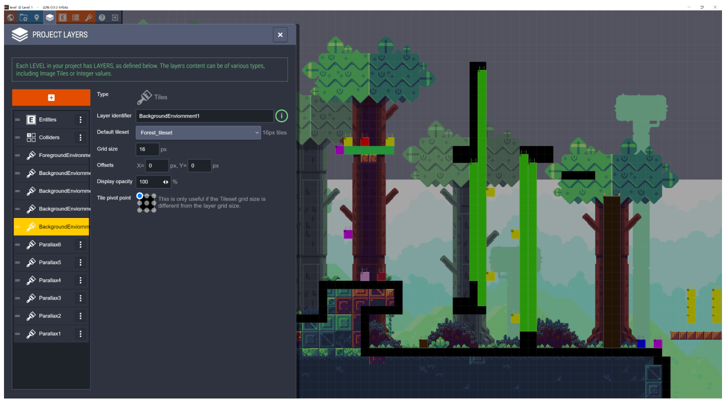
An upgrade has happened to the *TileGroup* class: it's now possible to specify what to do when a texture already exists on a given position with an enum called *BlendMode*. By using *BlendMode.Merge*, the new texture will draw every non-transparent pixel on top of the old texture. Similarly to opening 2 different sized images in an image editor, placing the small image onto the bigger one and saving the result merged into a new image file, but *TileGroup* also takes transparency into consideration to preserve the existing texture's pixels on every position where the new texture has no opaque pixels.
This way, it was enough to create just 1 entity for all background layers and 1 entity for all foreground layers. This has decreased the number of entities, and so the *Draw()* calls and Texture2D instances to 6: 4 created for the parallax layers, 1 for the background layers, and 1 for the foreground layers. Unfortunately, we can't merge all the parallax layers into one texture as their relative positions are changing with the camera movement due to the different scroll speeds.

Apart from constructing textures for environments, *TileGroups* are useful for all entities where the *Sprite* is composed of multiple textures or texture portions: by merging these together in a TileGroup, we can create one Texture2D instance to speed up the rendering.



Editing the map using the tile sheet (bottom left corner)


Visual representation of the entities placed on the map


List of foreground, background, and parallax layers parsed by the engine

2.13 UI

Implementation:

Engine/Source/UI/*.cs

User interfaces are important parts of video games: they allow the creation of menus and provide feedback for the players about how they are performing and in what state the game is.

The main properties of UI elements:

- They have no game logic attached, they usually just respond to input events and/or display different values
- Some of them do not scroll with the rest of the game, their position is fixed in the camera space

- They are not affected by the camera zoom level, although their scaling must be recalculated when changing display resolution
- They are running on their own, dedicated layer called *UILayer*
- They can be static or follow an entity (like a text bubble for the hero's thoughts)
- They can pause the game waiting for user input or disappear after a configured time
- They are always drawn on top of everything else, nothing can hide a UI element on the screen (except for another UI element)

Different type of UI elements implemented:

- Text output field (*TextField.cs*): not used in the demo game
- Image display (*Image.cs*): a static image, used when displaying the coin icon on the top left corner
- Selectable image (*SelectableImage.cs*): this can contain 2 images: an unselected and selected image, the appropriate one will be displayed based on whether the *IsSelected* flag is true.
- Selection list (*MultiSelectionImage.cs*): contains multiple images, the user can change the selection from a list of values, like scrolling through different screen resolution options.
- Static popups (*StaticPopup.cs*): a popup text and/or image that does not follow any entity and can scroll with the world. It can pause the game to wait for user input to continue or disappear after a timeout
- Dynamic popups (*DynamicPopup.cs*): scrolls with the world and can follow entities. They will disappear after a timeout.

Sounds effects can be attached to UI elements when applicable, for example, to play a sound when the mouse hovers over or the user clicks on the item.

2.14 Entry point of the game

Implementation: Engine/Source/Game/MonolithGame.cs

When creating a game with the engine, the entry class of the game must be a class that extends *MonolithGame*. This class does the initial configuration for the engine and MonoGame. By overriding the appropriate methods, the game developers can load all the assets they want and do any initial configuration.

2.15 Audio

Implementation: Engine/Source/Audio/AudioEngine.cs

The *AudioEngine* static class can be used to play audio files. It supports audio tags, currently 3 types: sound, music, and effects. The volume of these 3 categories can be adjusted individually and they can also be muted. The playback supports looping (background music, for example), pause and resume.

2.16 Assets

Implementation: Engine/Source/Asset/Assets.cs

The *Assets* is a static class that loads assets like images and audio files. It also caches them and if an asset is already loaded, it returns from the cache. A simple, convenient way to access all the assets from anywhere in the source code and reuse the existing instances for better performance.

3. Other noteworthy classes

3.1 Timer

Implementation: Engine/Source/Util/Timer.cs

An extremely useful static class that allows different timed events: set a timer for a property (like a cooldown), repeat an action for a period of time, or invoke a callback after a delay. It can also save us from writing a lot of boilerplate code in different classes where we want to code a time-driven behavior. It's updated in the normal *Update()* loop for high accuracy.

3.2 MathUtil

Implementation: Engine/Source/Util/MathUtil.cs

A small, static util class that contains the implementation of the most used math concepts/algorithms for 2D game development like geometry, algebra, etc.

3.3. Logger

Implementation: Engine/Source/Util/Logger.cs

A small, static class responsible for logging: it can log to the standard output as well as to a file using different log levels (info, debug, warning, error).

3.4 AssetUtil

Implementation: Engine/Source/Util/AssetUtil.cs

An internal static helper class to support loading and processing assets: it can load images, sounds and also auto-generate rectangular bounding boxes and circumscribed circles based on the images to use for rendering and physics purposes.

4. The future of the project

This engine isn't just my thesis work, it's also my personal project and I will keep working on it in the future. I would like to create a 2D video game engine that game developers love and use, and which brings joy to the player. I wouldn't just like to finish this engine, but I would like to use it to release the games that I'm planning, so there is still a long road ahead of me.
Two very important components are still to be implemented: object pooling and a particle system.

4.1 Object pooling

Creating new objects is very common during gameplay, but many of them are needed only for a short period of time, after which they will be garbage collected. But creating new objects and the garbage collector run have a performance impact, so instead of constantly creating and destroying objects, let's store them in a pool, from which we can take one if necessary and when we don't need it, we can put it back in the pool. This way, we can prevent frequent garbage collection and the potential performance impact on object creation.

4.2 Particle system

A particle system is a component responsible for displaying particles. Particles are usually groups of smaller sprites displayed in a similar manner, like raindrops, snowdrops, fragments/splinters leaving the center of an explosion, bullet shells, dust, smoke, etc...

Particles are usually not interactive and emitted using emitter classes. The way of emitting them is usually configurable: they can have a lifespan (after which they will get destroyed), the alpha value can be changed during the lifespan, gravity may or may not be applied to them, the shape and direction of the emission can be specified, their rotation and position can change with time, and most of these properties can usually be randomized to create believable particle effects. But because the count of the particles can be tens of thousands on the screen, the particle system must be written to aim for the highest possible performance, therefore it must be written from the ground-up as a separate component.

5. Acknowledgements

Creating a video game is a tremendous amount of work, even more so when it's being developed in parallel with the engine. I would like to thank Dr. Peter Bodnar for letting me choose this topic as my thesis. I've had an amazing time developing this code, the journey was very educational and it's far from over.

Furthermore, I would like to thank the whole independent video game developer community for the knowledge they keep sharing online, especially for:

- The MonoGame community for all the help they provide on their forums: https://community.monogame.net/
- Sébastien "deepnight" Benard for sharing decades of knowledge and work online with source codes and tutorials, and his community on his Discrod channel: https://deepnight.net/

6. Literature used for the work

- https://gafferongames.com/post/fix_your_timestep/
- https://deepnight.net/tutorials/
- https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm

Declaration

I, Lajos Rajna, hereby declare that my bachelor’s thesis was created at the University of Szeged, Faculty of Science and Informatics, Department of Image Processing and Computer Graphics to acquire my Computer Science bachelor’s degree. I also declare that this work has not been defended earlier in any other training program, it’s exclusively the result of my own work and I’ve used only the referenced sources (literature, tools, etc.). I acknowledge that my bachelor’s thesis will be placed in the library of the University of Szeged, Faculty of Science and Informatics among the books to read in place.

2021.05.16

Lajos Rajna
Signature