



Game Programming

Lecture 5: Visual effects & sound effects

January 16, 2023

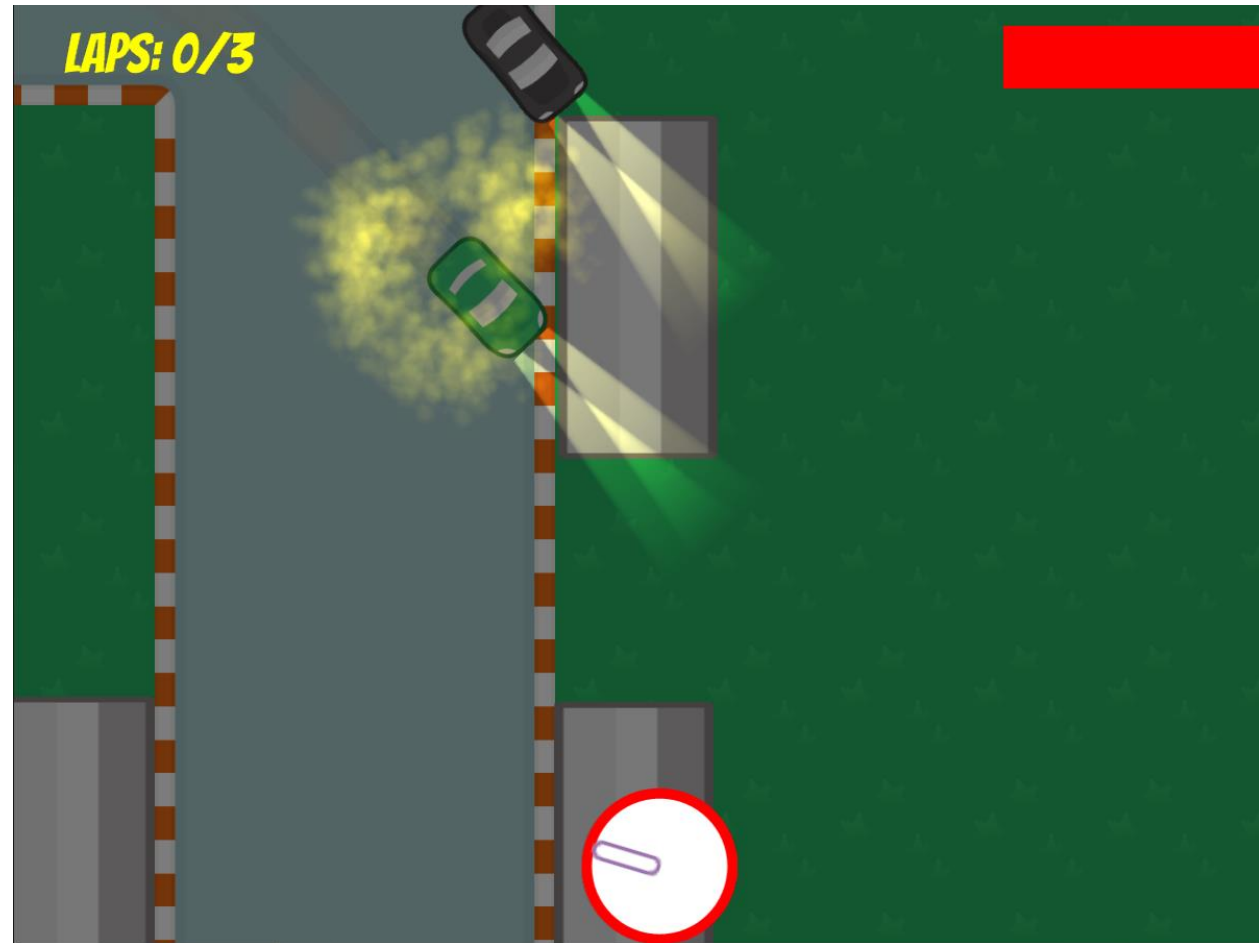
Paul Bonsma



Previous Weeks

- Previous weeks:
 - Getting started: creating a movable, animated sprite
 - Game objects, interaction, collisions
 - Level loading
 - Creating a user interface
 - Code architecture (inheritance)
- You should (could?) have a basic playable game now, with a clear user interface
- ...but how does the *game feel*?

Last Week vs This Week (Demo)



Improvements

What kind of improvements to the “*game feel*” do you see?

- Dynamic sounds (engine)
- Trails (tire tracks)
- Particles (exhaust smoke & explosions)
- Screen shake (on impact)
- Lighting (head lights)



→ Aspects like these are also called “*juice*”, and are very common in modern games (essential to make professional impression!)

(Strategy, challenge, mechanics, progression → appeal to *rational* brain functions.

Story, narrative, art, music → appeal to the *emotional* side

Juice, *explosions*, *particles*, *movement* → entertain the *lizard brain* 😊)

User Feedback / Juice

- “Juice it or lose it” (Martin Jonasson & Petri Purho):
<https://www.youtube.com/watch?v=Fy0aCDmgnxg>
- “The art of the screen shake” (Jan Willem Nijman):
<https://www.youtube.com/watch?v=AJdEqssNZ-U>
 - <https://youtu.be/AJdEqssNZ-U?t=442> (start)
 - <https://youtu.be/AJdEqssNZ-U?t=1629> (end)
- “Secrets of Game Feel and Juice” (Game Maker’s Toolkit):
https://www.youtube.com/watch?v=216_5nu4aVQ
- Example game (Nuclear Throne): <https://youtu.be/7LSs1bj41P4?t=48>

...interested?

Lecture Outline

- (Mathematical) functions (yes, it all starts with some math...)
- Sound
- Blend modes & lighting effects
- Particles
- Tweening
- Conclusions

Don't Stress

- *To pass this course, you only need* to add sound to your game (which is pretty easy)
- The other information just *helps* to...
 - ...get a higher grade
 - ...be able to create something awesome in the upcoming projects
 - ...become a skilled (game) programmer!

Functions

Mathematical Functions

- A (mathematical) *function* is something that maps an *input* variable to an *output* variable
- These variables can be anything (vectors, colors, rotations, ...), but we focus on one-dimensional functions: float \rightarrow float.
- Typical notation: x = input, y = output. Function f : $y = f(x)$
- Examples: Code:
 - $y = 2x + 5$ `y = 2*x + 5;`
 - $y = x^2$ `y = x*x; y = Mathf.Pow(x, 2);`
 - $y = \sqrt{x}$ `y = Mathf.Sqrt(x);`

Why?

Examples:

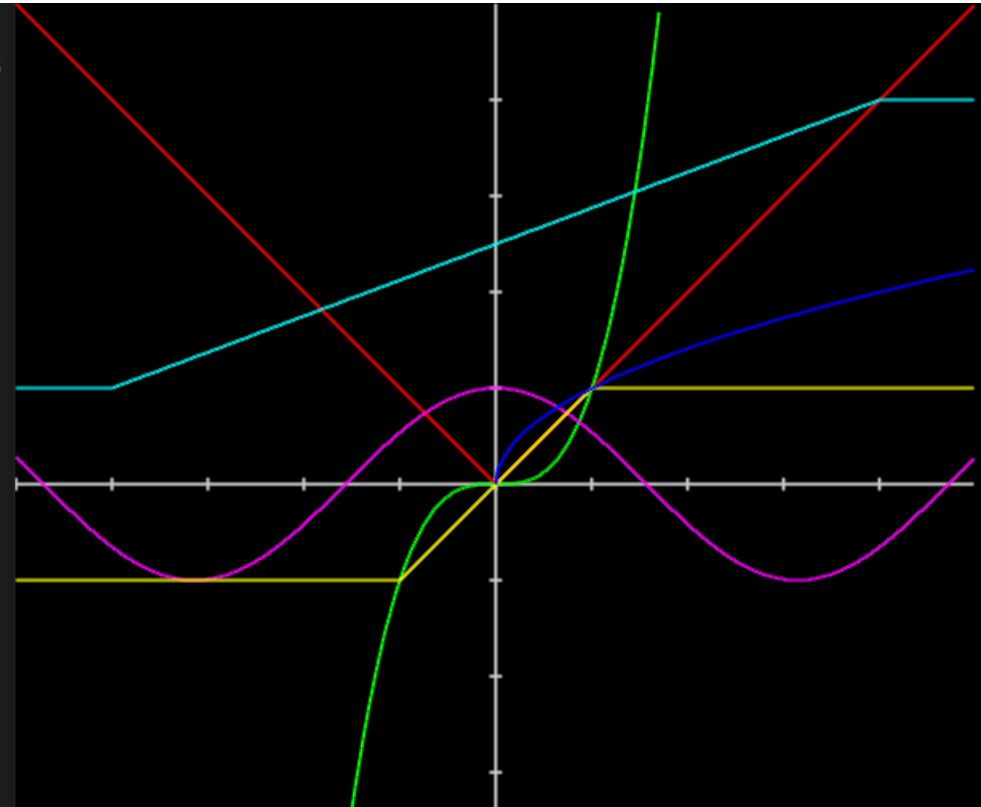
- Directional sound: map screen coordinates to audio volume & panning
- Engine sound: map car speed to sound frequency
- Tweening (animation) and particles: map time to position / rotation / alpha...
- (Understanding blend modes)
- ...and that's just this lecture!

Plotting functions

- On Blackboard you can find a function *plotter* project that can help you *learn about / explore / design* functions

```
// example functions:  
1 reference  
float Cubic(float x) {  
    return x * x * x;  
}  
  
1 reference  
float MapExample(float x) {  
    return Map(x, -4, 4, 1, 4, true);  
}
```

```
1 reference  
public MyGame() : base(800, 600, false)    // Create a window that's  
{  
    Plotter plot = new Plotter(600, 600, -5, 5, -5, 5);  
    AddChild(plot);  
    plot.DrawAxes();  
    plot.StrokeWeight(2);  
    plot.Stroke(255, 0, 0);  
    plot.Plot(Mathf.Abs); // Red: absolute value  
    plot.Stroke(0, 255, 0);  
    plot.Plot(Cubic); // green: cubic function (custom)  
    plot.Stroke(255, 0, 255);  
    plot.Plot(Mathf.Cos); // magenta: cosine  
    plot.Stroke(255, 255, 0);  
    plot.Plot(ClampFixed); // yellow: clamp between -1 and 1  
    plot.Stroke(0, 0, 255);  
    plot.Plot(Mathf.Sqrt); // blue: square root  
    plot.Stroke(0, 255, 255);  
    plot.Plot(MapExample); // cyan: map from -4..4 to 1..4 (clamped)  
}
```



Mathf

- Mathf contains many useful functions
- Make sure you understand these (see comments in the code!):
 - Abs, Sign
 - Round, Ceiling, Floor
 - Min, Max, Clamp
 - Pow, Sqrt
- Later (Physics Programming):
 - Sin, Cos, Tan
 - Atan, Acos

Quiz: Creating Linear Functions (Map)

- Map 0..1 to 0..**10**?
 - $y = 10 * x$
- Map 0..1 to **4**..14?
 - $y = 4 + 10 * x$
- Map 0..**5** to **4**..14?
 - $y = 4 + 10 * (x/5) = 4 + 2 * x$
- Map **10**..15 to **4**..14?
 - $y = 4 + 10 * (x-10)/5 = 2 * x - 16$

Linear Interpolation / Map

Two very useful formulas:

- *Linear Interpolation*: map a value t between $0..1$ to the range $\text{minOutput}..\text{maxOutput}$

```
return outputMin + t * (outputMax - outputMin); // map 0..1 to outputMin..outputMax (=linear interpolation formula)
```

- *Linear mapping*: map a value x between $\text{minInput}..\text{maxInput}$ to the range $\text{minOutput}..\text{maxOutput}$

```
/// <summary>
/// Linearly map a value x between inputMin..inputMax to a value between outputMin..outputMax
/// </summary>
1 reference
float Map(float x, float inputMin, float inputMax, float outputMin, float outputMax, bool clamp=false) {
    float t = (x - inputMin) / (inputMax - inputMin); // map values between inputMin..inputMax to 0..1
    if (clamp) { t = Mathf.Clamp(t, 0, 1); }
    return outputMin + t * (outputMax - outputMin); // map 0..1 to outputMin..outputMax (=linear interpolation formula)
}
```

Sounds

Sounds

- You can create a *Sound* object from a sound file (format: wav, mp3, ogg)
- *Looping*: At the end of the sample, it will play from start again, until stopped
- *Streaming*: The file is not loaded into memory, but dynamically loaded from disc (use this for large music tracks)
- Call the *Play* method to play the sound (returns a *SoundChannel*)

```
engine = new Sound("engine.ogg",true,false).Play();
```

🔗 `Sound.Sound(string filename, [bool looping = false], [bool streaming = false])`

Creates a new `Sound`. This class represents a sound file. Sound files are loaded into memory unless you set them to 'streamed'. An optional parameter allows you to create a looping sound.

Sound Channel

- mySound.Play() returns a *SoundChannel*
- Using this, you can:
 - *stop, pause, mute* the sound
 - Change *volume, pan, frequency* (=pitch and playback speed)
 - Volume: between 0 (=silent) and 1 (=full volume)
 - Pan: between -1 (=left) and 1 (=right). 0 = center
 - Frequency: strictly positive. 44100 is the default frequency (=CD quality).

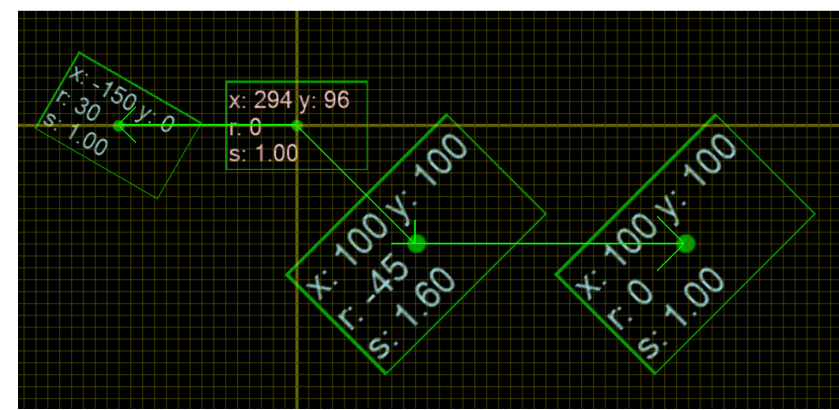
```
void SFX() {  
    //float pan, volume;  
    float pan = PanPosition();  
    float volume = VolumePosition();  
    engine.Pan = pan;  
    engine.Volume = volume;  
    // Linear mapping: speed=4 maps to 44100 (=standard frequency):  
    engine.Frequency = 44100 * (Mathf.Abs(speed)+1) / 5;  
}
```

Example Applications

- *Directional sound*: change pan and volume depending on screen position
- *Engine sound*: slow = low pitch, fast = high pitch
- *Dynamic, adaptive music*:
 - Play at least two tracks (with same length?) in sync
 - Fade them in and out / crossfade them, depending on game events
- (Demo's)

```
float PanPosition() {  
    // Compute the screen position for this sprite's origin point  
    // (requires using GXPEngine.Core namespace):  
    Vector2 screenPosition = TransformPoint(0, 0);  
    // Linearly map screen positions between 0..game.width to pan between -1..1 (clamped):  
    return Mathf.Clamp(  
        2 * screenPosition.x / game.width - 1,  
        -1, 1  
    );  
}
```

(Inverse)TransformPoint



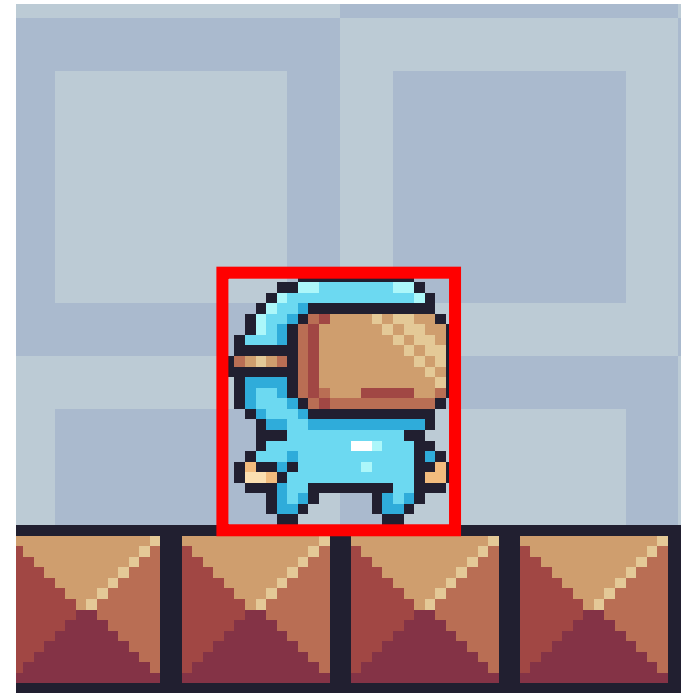
- In any GameObject, the *TransformPoint* method translates a point in *local space* to *screen space*
- *InverseTransformPoint* translates a point from screen space to local space
- Returns a Vector2 → requires GXPEngine.Core namespace

```
// Here's how you can create a game object p at this object's position, but
// make it child of the parent's parent:
// (Note: make sure parent and parent.parent aren't null!)
var scrPos = TransformPoint(0, 0); // transform our origin to screen space
var pos = parent.parent.InverseTransformPoint(scrPos.x,scrPos.y); // transform to parent-of-parent space
p.SetXY(pos.x, pos.y);
parent.parent.AddChild(p);
```

Blend Modes

Transparency

- Recall:
 - every sprite *pixel* has a *color* (red, green, blue values), but also a *transparency* (=alpha value)
 - Alpha = 0 means fully transparent (invisible), alpha = 1 means fully opaque
 - *Sprite objects* also have color and alpha values: pixel colors are multiplied with these values
 - So you can only make the pixel *darker* or *more transparent* with these values
- Using alpha values this way to “transparently blend” sprites with previously rendered sprites is the default, but not the only way! → **BlendModes**

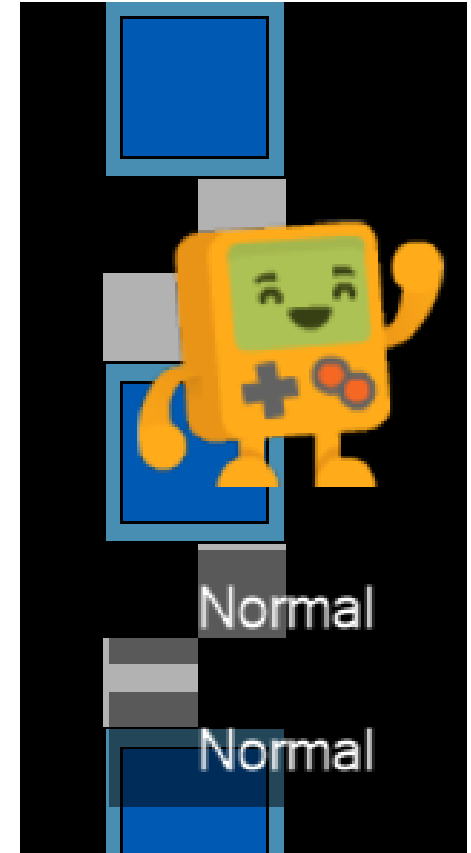


Normal BlendMode

- For every pixel that is rendered:
 - New color = sprite pixel color * alpha + previous color * (1-alpha)

```
/// <summary>
/// The traditional and default way of blending.
/// (newColor = spriteColor * spriteAlpha + oldColor * (1-spriteAlpha))
/// </summary>
public static readonly BlendMode NORMAL = new BlendMode (
    "Normal", () => { GL.BlendFunc(GL.SRC_ALPHA, GL.ONE_MINUS_SRC_ALPHA); }
);
```

Hey! There's our friend linear interpolation again!



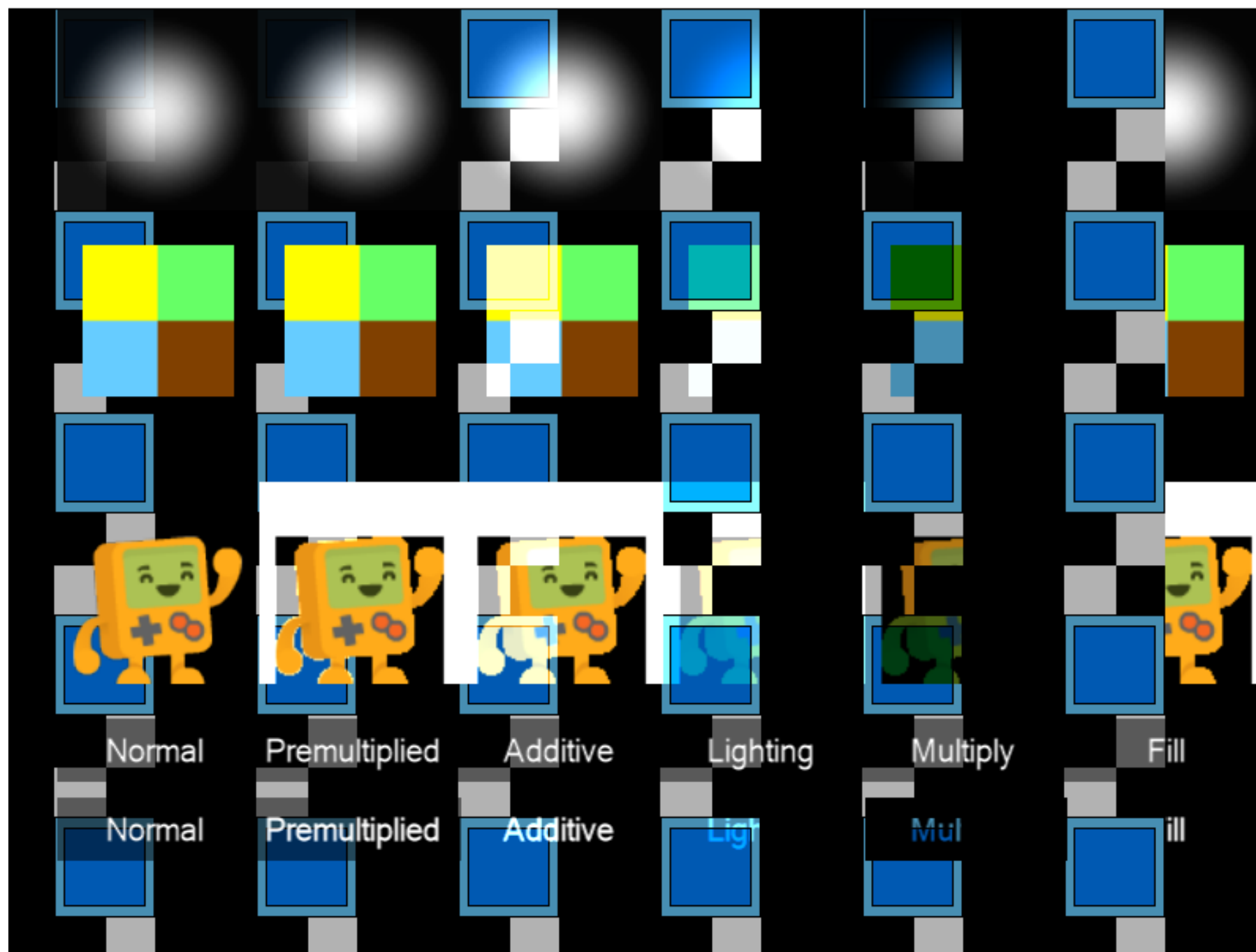
- This is the default, but there are other blend modes!
- See [Core/BlendMode.cs](#) and the blend mode example on Blackboard

```
/// <summary>
/// Multiplying colors - use this for darkening.
/// (newColor = spriteColor * oldColor + oldColor * 0)
/// </summary>
public static readonly BlendMode MULTIPLY = new BlendMode (
    "Multiply", () => { GL.BlendFunc(GL.DST_COLOR, GL.ZERO); }
);

/// <summary>
/// Brightening existing colors - this mode can be used for lighting effects.
/// (newColor = spriteColor * oldColor + oldColor * 1)
/// </summary>
public static readonly BlendMode LIGHTING = new BlendMode(
    "Lighting", () => { GL.BlendFunc(GL.DST_COLOR, GL.ONE); }
);

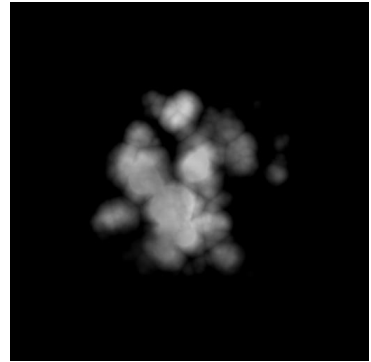
/// <summary>
/// Adding colors - use this e.g. for "volumetric" lighting effects.
/// (newColor = spriteColor * 1 + oldColor * 1)
/// </summary>
public static readonly BlendMode ADDITIVE = new BlendMode(
    "Additive", () => { GL.BlendFunc(GL.ONE, GL.ONE); }
);

/// <summary>
/// This mode can be used to fill in empty screen parts (e.g. drawing a background after adding lights to the foreground)
/// (newColor = spriteColor * (1-oldColorAlpha) + oldColor * oldColorAlpha)
/// </summary>
public static readonly BlendMode FILLEMPY = new BlendMode(
    "Fill", () => { GL.BlendFunc(GL.ONE_MINUS_DST_ALPHA, GL.DST_ALPHA); }
);
```



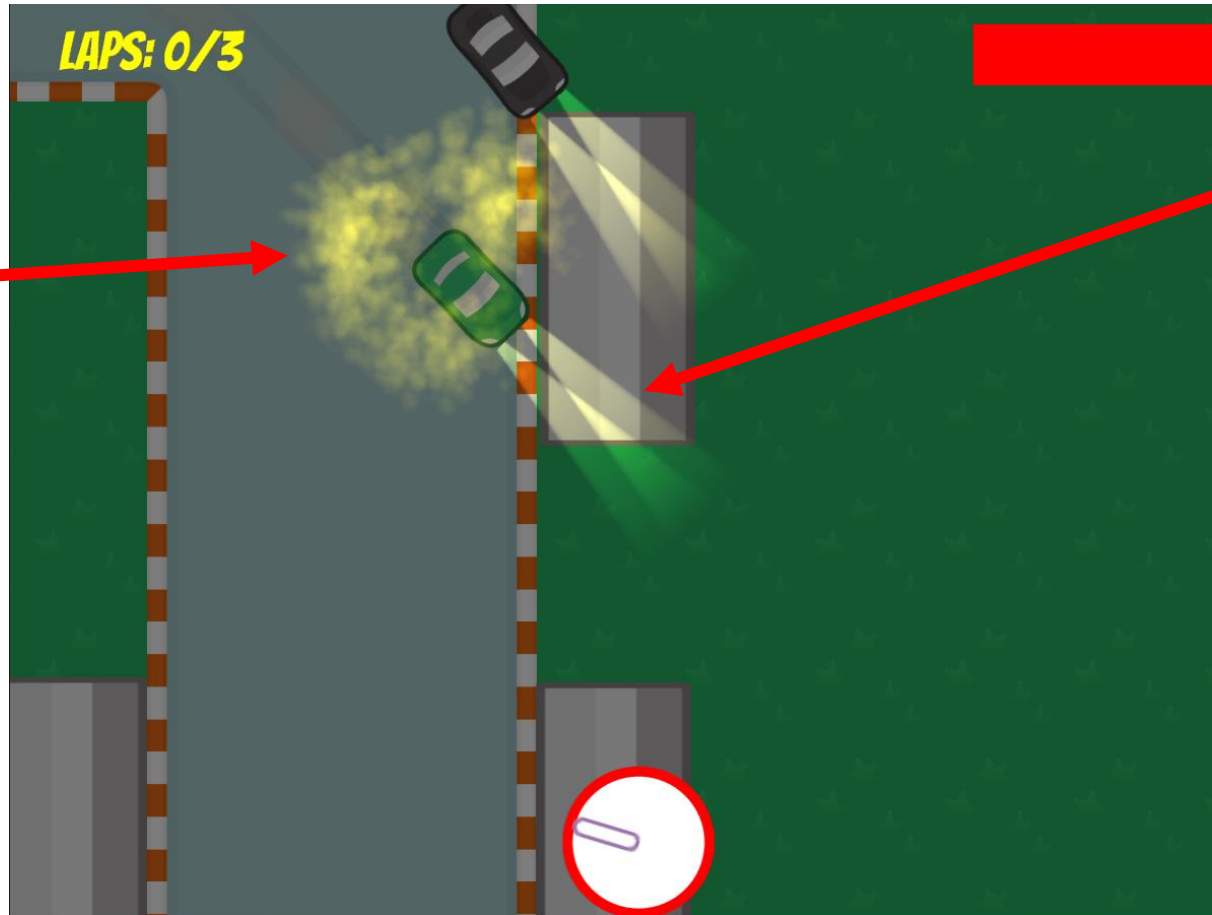
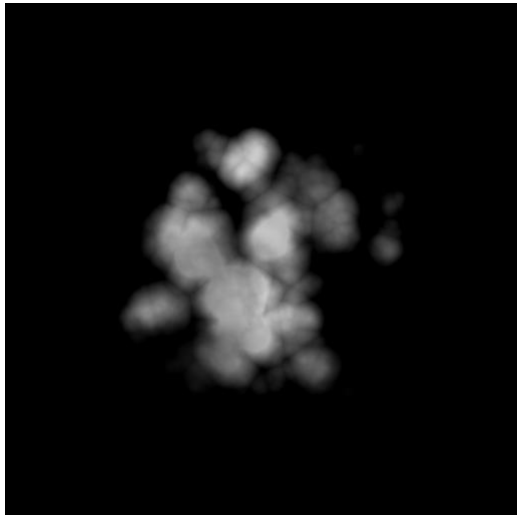
Example Use Cases

- Blend modes where alpha values are ignored (typically: use grayscale sprites):
 - *Additive*: Often used for particles.
 - *Lighting*: Use this with “light cookie” sprites to light up dark sprites
 - *Multiply*: Use this to “darken” previously rendered sprites
- Using the previous pixel’s alpha(!):
 - *FillEmpty*: after applying lighting (see above), draw an unlit background (note: this doesn’t work well using sprites with pixel alpha values other than 0 or 1, so use pixel art sprites for this, or `BlendMode.Premultiplied`)



GXP Engine Example

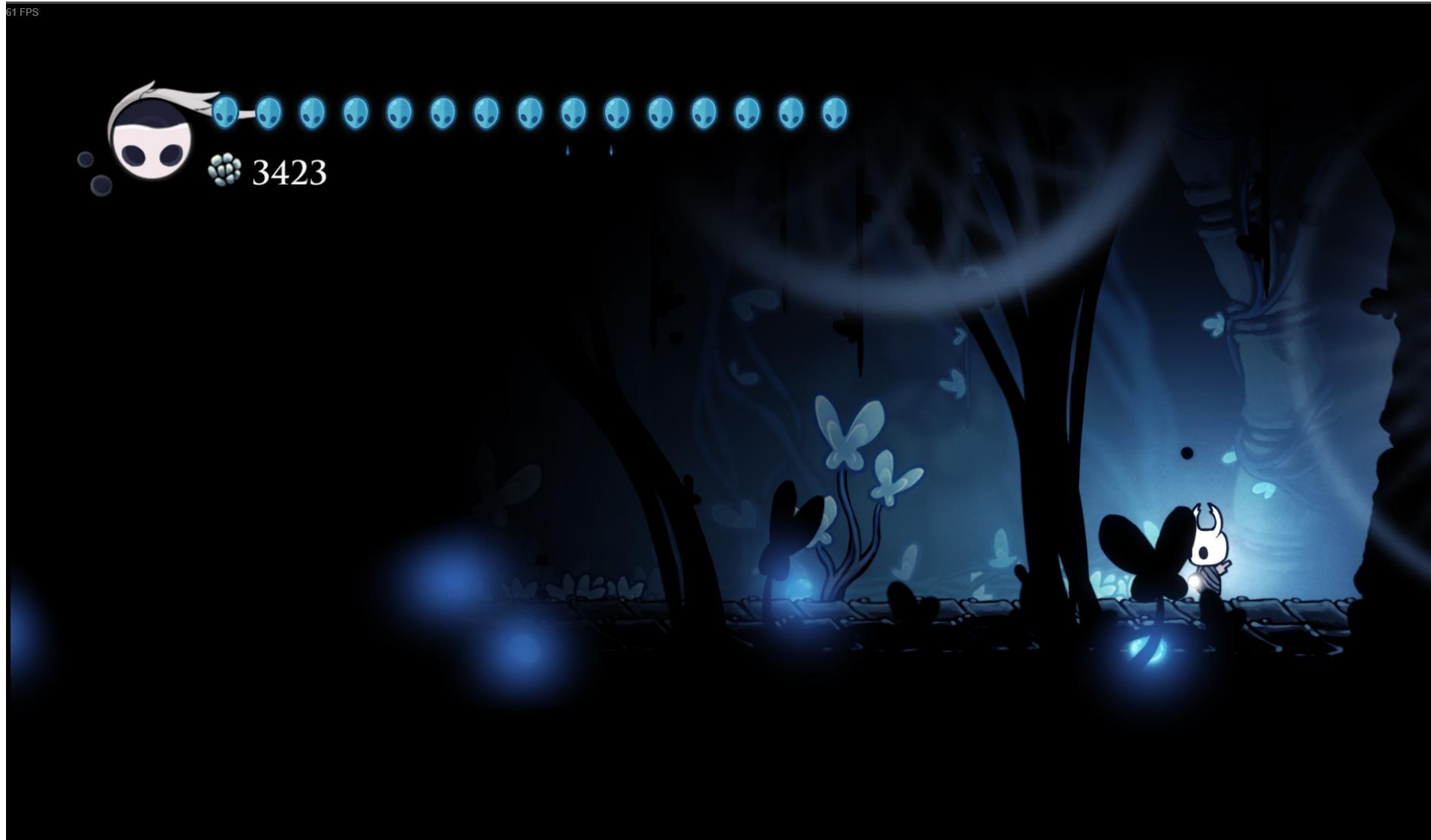
BlendMode.ADDITIVE



BlendMode.LIGHTING

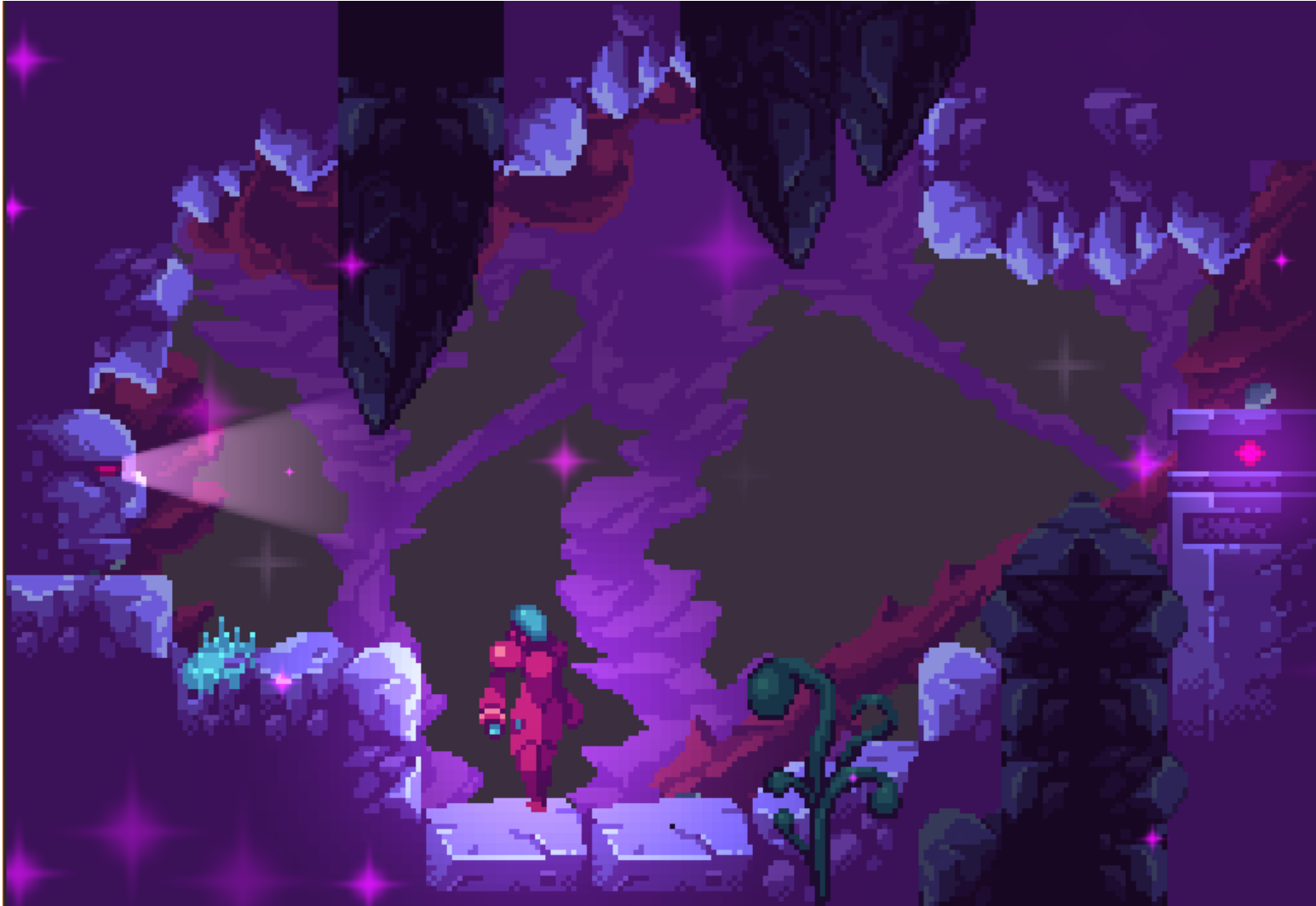


2D Lighting Case Study: Hollow Knight



How can you
create this look?

2D Lighting Example (GXPEngine)



Assets:

- Mostly from **Ansimuz** (*warped* - opengameart.org)
- Some Kenney (kenney.nl)
- Some self made (lights)

Render order & BlendMode:

1. Lit BG & main layer
2. Lights (BM: *Lighting*)
3. Background (BM: *FillEmpty*)
4. Volumetric lights (BM: *Add*)
5. Foreground tiles
6. Vignette (BM: *Multiply*)
7. Particles (BM: *Add*)

See the *handout on Blackboard* for details

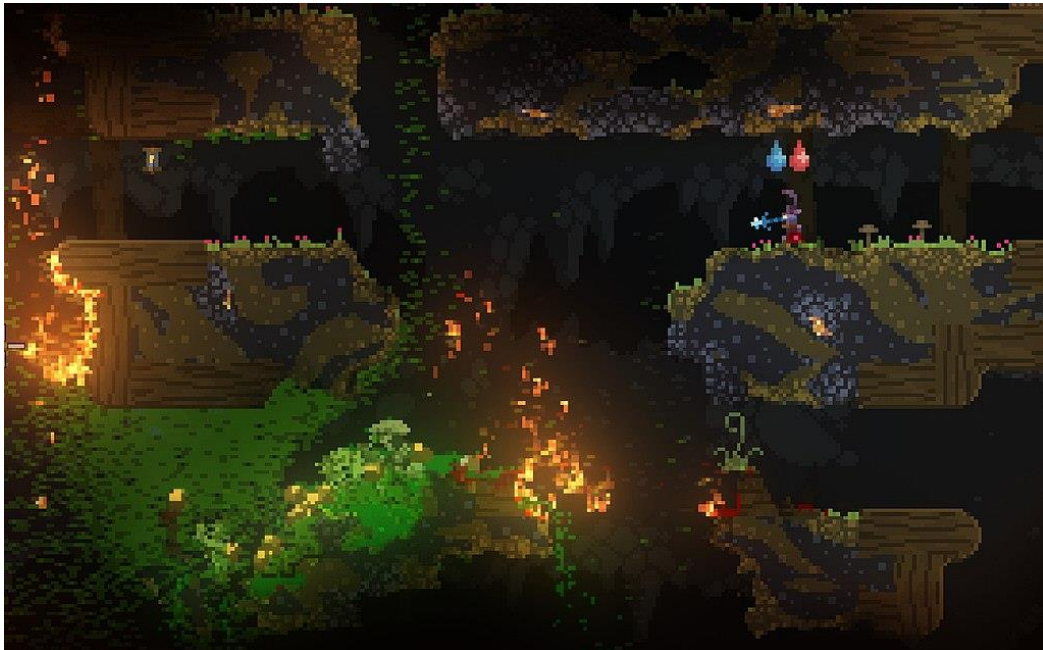
Particle Effects

Particles

- *Particles* are moving sprites used for visual feedback or decoration (no game play functionality), usually found in large groups
- They are *everywhere* in modern games! Examples:
 - Fire
 - Smoke
 - Weather (rain, snow)
 - Spell casting / magic
 - Shooting a gun: dropping shells
 - Walking: dust clouds
 - Showing collision impact
 - Breaking into parts
 - EXPLOSIONS!

Particle Effects - Examples

- Noita - The king of particle effects...: <https://www.youtube.com/watch?v=ZBLoffoZLH8>
- Destructivator 2: https://www.youtube.com/watch?v=Q5_RN335mck
- Made in the GXP Engine: <https://youtu.be/2wID0vNPFjM?t=227>
(Ricatchet – Project Liftoff 2020)



Basic Particle Example (GXPEngine)

- See the *handout on Blackboard* for details: it contains a basic *Particle class* that you can use as starting point

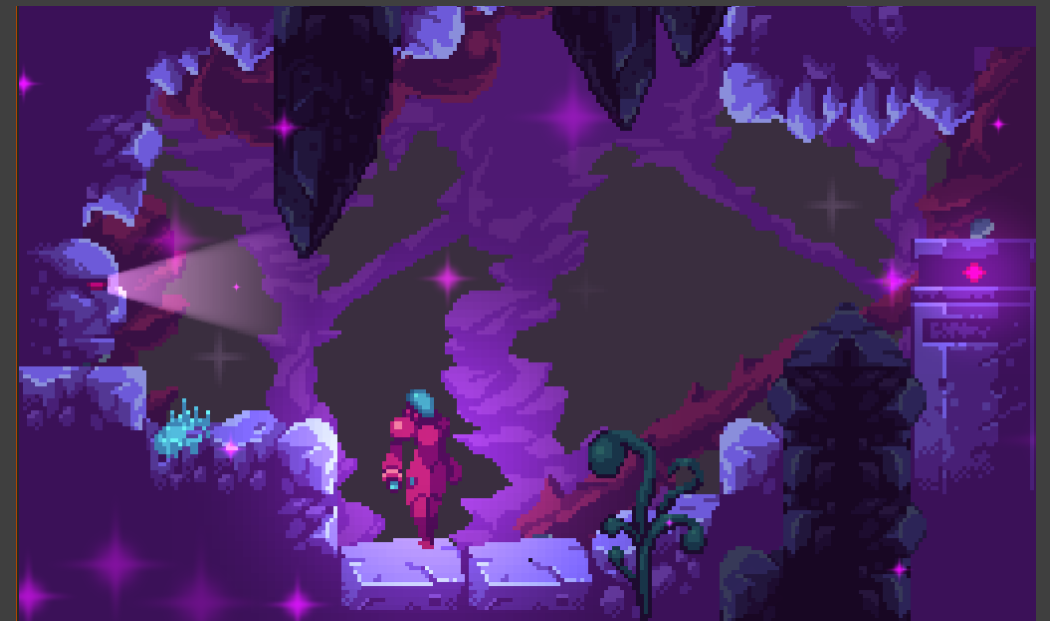
```
void Update() {
    currentLifeTimeMs += Time.deltaTime;
    // Make a parameter that goes from 0 to 1 throughout the lifetime of the particle:
    float t = Mathf.Clamp(1f * currentLifeTimeMs / totalLifeTimeMs, 0, 1);

    // ---- Interpolate parameters, using linear interpolation (you can use tweening curve)
    // interpolate color:
    float currentR = startColor.R * (1 - t) + endColor.R * t;
    float currentG = startColor.G * (1 - t) + endColor.G * t;
    float currentB = startColor.B * (1 - t) + endColor.B * t;
    SetColor(currentR/255f, currentG/255f, currentB/255f);

    // interpolate scale:
    scale = startScale * (1 - t) + endScale * t;

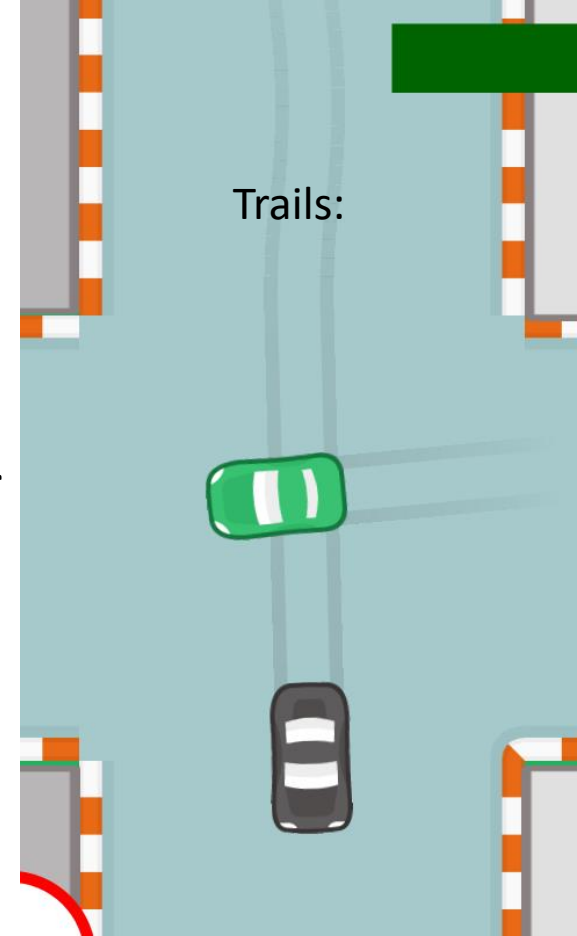
    // Move:
    x += vx;
    y += vy;

    if (currentLifeTimeMs >= totalLifeTimeMs) {
        Destroy();
    }
}
```



Code Setup Tips: Particle

- Create a *Particle* class, which is a sprite
- Every particle has a *lifetime* (say 5 seconds) and destroys itself afterwards (use `Time.time` or `Time.deltaTime`)
- In Update:
 - Over its lifetime, a parameter t increases from 0 to 1
 - Use *linear interpolation* (with parameter t) to change over time:
 - Rotation
 - Scale
 - Color
 - Alpha
 - Change the position using x- and y-*velocity* values (optionally: the velocity changes over time, influenced by gravity)
- Example: *trails* (tire tracks in the race game example) are particles – every frame the car creates one
- See the code sample on Blackboard for a simple example




Particle sprite:



Particle Emitter

- A particle *emitter* emits particles, for instance:
 - Constant rate, during its lifetime (example: fire)
 - A short burst of particles (example: explosion)
- Emitter properties:
 - All the properties of its particles
 - Start & end color
 - Start & end scale
 - Blend mode
 - Particle lifetime
 - ...
 - Randomizer properties:
 - Possible spawn positions (a single point, or a rectangle, or within a circle, ...)
 - Possible velocity values
 - Scale randomizer range, rotation randomizer range, ...
 - ...



That's *a lot* of properties!

Code Setup Tips: Emitter

- Create an *Emitter* class (GameObject)
- Constructor: essential properties (image file, blend mode, ...)
- Add methods to set other properties
- These methods return *this*, to allow for 'chaining'!

```
// ----- Snow effect:  
// Create 10 particles/second, lifetime = 10 seconds:  
emit = new Emitter("snowflake.png", 1, 1, 10, 10, BlendMode.NORMAL);  
// movement angle between 60 and 120 degrees, speed between 50 and 100 pixels/second:  
emit.SetVelocity(60, 120, 50, 100).  
    // Color white, but slightly transparent (alpha=0.6) - no change over time:  
    SetColor(1, 1, 1, 1, 1, 1, 0.6f, 0.6f).  
    // Scale doesn't change, but randomized between 0.2 and 1 (=0.2+0.8):  
    SetScale(0.2f, 0.2f, 0.8f).  
    // particles spawn in the range x=0..1280, y=0 (=top of screen):  
    SetSpawnPosition(1280, 0);
```

```
public Emitter SetVelocity(float minAngle, float maxAngle, float minSpeedPps, float maxSpeedPps) {  
    this.minAngle = minAngle;  
    this.maxAngle = maxAngle;  
    minSpeed = minSpeedPps;  
    maxSpeed = maxSpeedPps;  
    return this;  
}
```



Demo



Creating Particle Effects - Summary

- With these tips + your current knowledge, you can create your own *particle effects tools*
 - Sprite color, alpha, position, rotation, scale
 - Game objects, Update, lifetime (Time.time)
 - Linear interpolation
 - Code setup (Particle / Emitter)
- Don't forget to create “design tooling”, where you can quickly tweak all these parameters, and do your visual design (sliders?)
- You can make it as simple or complex as you want (creating one specific effect is easy – creating versatile tooling is more work)

Efficiency

- Don't forget: particles shouldn't have colliders! (Usually)
- This setup where every particle is a game object is easy, and sufficient in many cases, but not efficient enough when you need a huge number of particles
- For Noita-level effects: a data-oriented architecture is needed, or possibly compute shaders → outside scope of this course!

Tweneing

Animation by Code: Tweening

- Similar to particles, you can `animate` all kinds of game object *properties* by code, for *any game object*
 - Change position / rotation / scale over time
 - Change color / alpha over time
- Example: `move this game object 200 pixels to the right during 1 second`
- This is often used for UI elements (just look at almost any modern game...)
- You can also do this for other game objects (e.g. grabbed pickups or killed enemies fade out over time)
- This is called '***Tweening***' ('tween' is short for inbetween)
- Key feature to make it look good: don't use linear interpolation – use ***tweening curves*** (key word: *easing*)

Tweening in Action

- <https://youtu.be/Fy0aCDmgnxg?t=256> (From “Juice it or Lose it”)

Example Curves



Source: easing.net

Code Setup Tips: Tweening

- Create a *Property* enum with all the game object / sprite properties you may want to animate

```
public enum Property { x, y, rotation, scale, alpha, color }
```
- Create a *Curves* enum with all the possible curve types (e.g. Curves.Linear, Curves.EaseInOut, Curves.EaseOutBack, ...)
- Create a *Tween* class (game object), with as parameters:
 - Target game object (or sprite)
 - The property that should be animated
 - Lifetime
 - Amount of change (Move 300 pixels to the right in 1.5 second)
 - Curve

```
sprite.AddChild(new Tween(Property.x, 1.5f, 300, Curves.EaseOutBack));
```
- In Update, a Tween object animates its target (as expected)
- *Actually*, it's easier to choose the *parent* game object as default target!
- Make sure a tween object destroys itself when it's done!

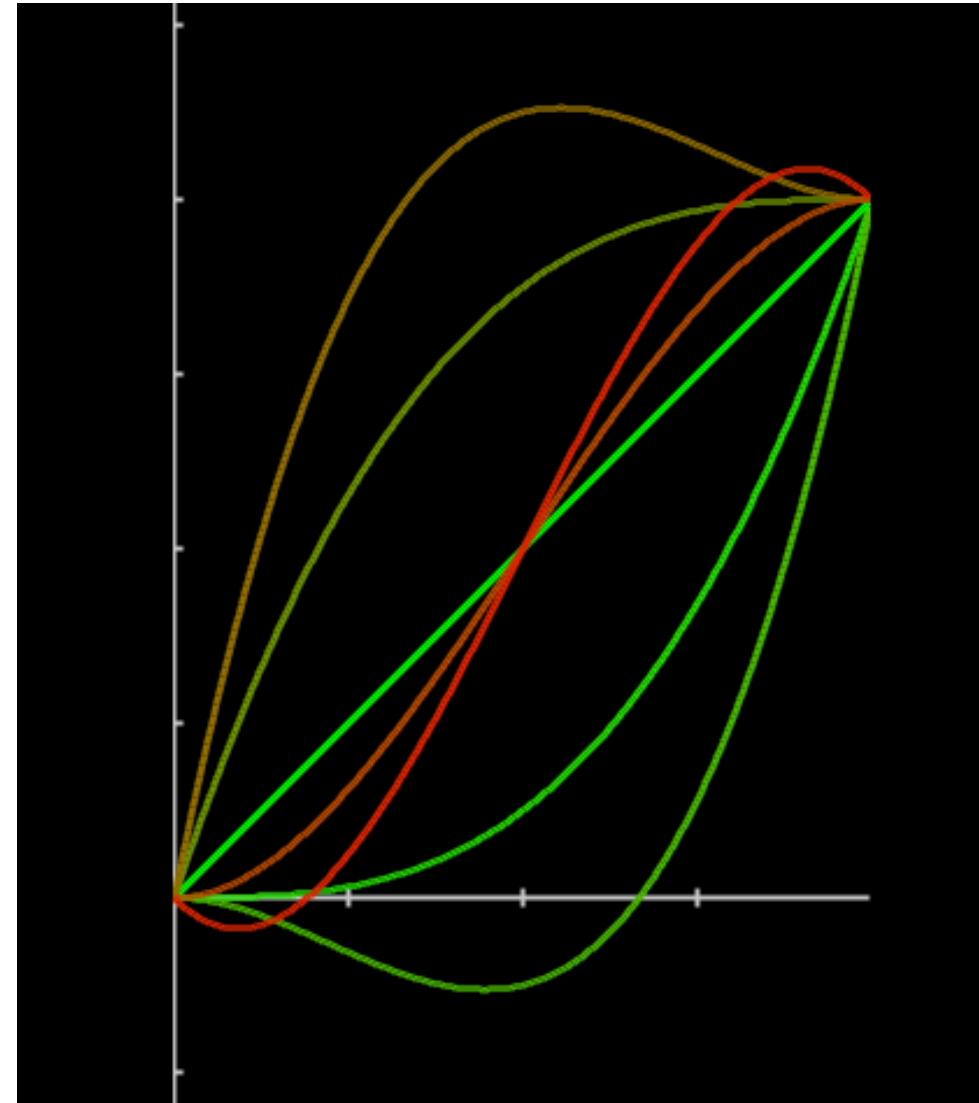
Demo

- (Animating different properties, different curves)

Example Curves: Formulas

In your program: every curve is a function f with input range $0..1$ and typical output range $0..1$, often with $f(0)=0$ and $f(1)=1$

Name:	Formula:
Linear	$y = x$
EaseIn	$y = x^3$
EaseInBack	$y = 3x^3 - 2x^2$
EaseOut	$y = x^3 - 3x^2 + 3x$
EaseOutBack	$y = -1.5x^3 + x^2 + 1.5x$
EaseInOut	$y = -2x^3 + 3x^2$
EaseInOutBack	$y = -4x^3 + 6x^2 - x$



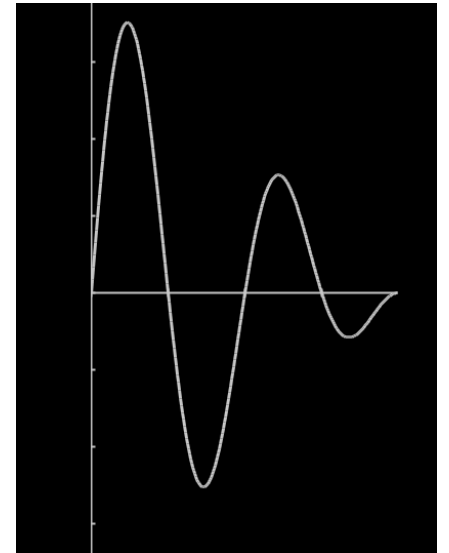
Creating Curves

- Using the *Plotter* handout (see Blackboard), you can create and view your own custom tweening curves
- For inspiration, see <https://www.youtube.com/watch?v=mr5xkf6zSzk> (Math for Game Programmers: Fast and Funky 1D Nonlinear Transformations – Squirrel Eiserloh, GDC talk 2015)
- You can also create curves that start and end at zero (e.g. for screen shake)

Highly recommended!

```
float SinDamp(float t) {  
    return Mathf.Sin(Mathf.PI * 4 * t) * (1-t);  
}
```

```
// one line screen shake:  
game.LateAddChild(new Tween(Property.x, 0.7f, 3 * Mathf.Abs(speed), Curves.SinDamp));
```



Conclusions

Summary

- Game feel / juice: how and why
- (One dimensional) functions, basics
- Playing sounds + sound channel properties for dynamic sound
- Blend modes & lighting effects
- Particles, emitters
- Tweening, curves

Why Like This?

Q: Why are the tools for particles / tweening / dynamic music not simply built into the GXP Engine? That would help us to make awesome games...

Recall:

- The goal is not for us to give you a powerful engine and then for you to *learn how to use all the built-in tools* (that will happen with Unity)
- The goal is for you to *become a good game programmer*. You can only do that by *practicing*. Creating your own particle / tweening / dynamic music tooling is a great exercise!
- It's not necessary to pass this course, but it will help for the upcoming projects (by the way: if you do it well, you can later reuse your tweening + dynamic music tooling in Unity!)

A Look at the Grading Criteria

user feedback (20 pt)	-∞ pt	12 pt	16 pt	20 pt
	<p>The student shows insufficient control of implementing user feedback. The project lacks a clear HUD, suitable sprites or sound effects.</p>	<p>The student has implemented necessary user feedback (HUD, animated sprites, sound effects) to the game.</p>	<p>The student has implemented more than basic user feedback features (e.g. effective use of particle effects, sound pitch, volume and pan, blend modes or lighting)</p>	<p>The student shows excellent control of implementing user feedback by making a game that provides seamless feedback to the player. The game feels very responsive.</p>

This Week / During the Lab

- Add sounds to your game
- Getting sounds:
 - <https://opengameart.org> / <https://freesound.org>
 - Recording some sounds yourself is possible too. You can edit them using e.g. <https://audiomass.co>

Optional:

- Create (tooling for) particle effects / tweening effects / dynamic music / directional sounds / lighting effects (not all of these - pick your favorite!)
- You can make this as simple or complex as you want
- Give and receive some *code reviews*!

Code Review

- A code review form will be posted on Blackboard
- During the lab, in pairs:
 - show and discuss your code, get feedback
 - To give feedback: fill in the form

code (20 pt)	-∞ pt	12 pt	16 pt	20 pt
	The student shows insufficient control of maintaining code quality by writing code that doesn't use the course prescribed coding conventions, having many long methods, or excessive code repetition.	The student writes code that follows that prescribed coding conventions , using clear names and adding comments where applicable. (Most) methods have a clear, single task and are no longer than one page. There is limited repeated code .	The students proves that the code is reviewed on readability and code standards and by a teacher and at least two peers. The student explains how their remarks were used to establish or improve the code quality.	The student proves substantial effort to write stable code and prevent runtime bugs .