A documentation showcasing the game logic and thought process

Tiago Santos Silva

WEMOB2

B1GP

Table of Contents

Engine

For the JavaScript engine, I chose Kaplay.js. This is the newer and rebranded version of Kaboom.js.

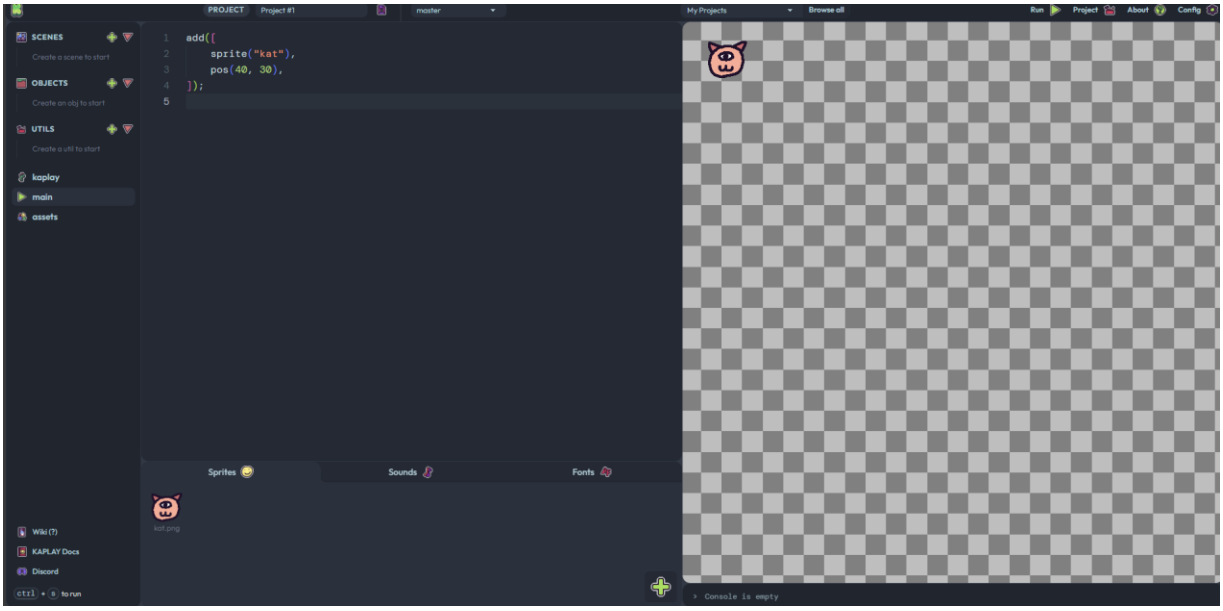
The reason for this choice is that I found that the engine has a very straightforward library and syntax, utilizing gaming terminology all across the board while keeping it fairly simple. Not to say that you can't do complex things with this, of course.

The way I went about programming using it, is that I operated strictly on their main website, due to it having the newest version of Kaplay. When downloading it to your device, it downloads the current main version, while their website uses the newest version 4 with some additional bonuses with it. Due to this, the only way to have the game properly running while being able to code, is on their website:

https://play.kaplayjs.com

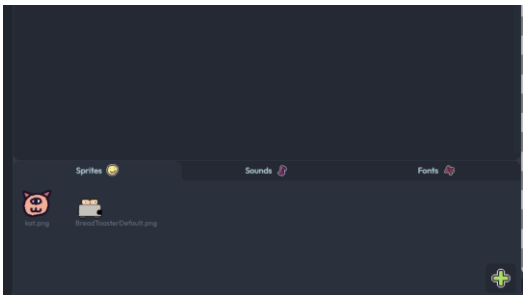
Setup

When opening this link, you'll be welcomed with this screen.



Simply copy the code I have sent inside the **index.js** file and paste it in this exact file that automatically should open. (If it doesn't, it's the "main" file that you can see on the left side of the screen.)

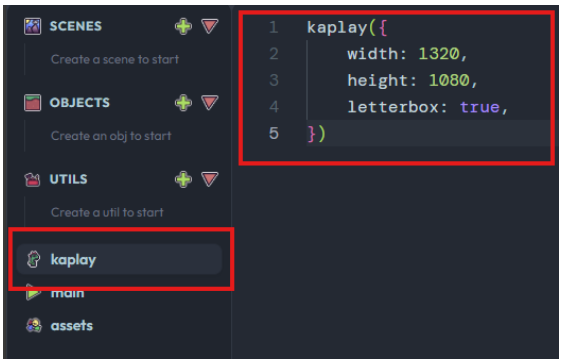
Next, make sure to add all the art that can be found inside the "Art" folder of the file I have submitted. You can do it by drag and drop of the images into the Sprites section at the bottom of the screen or click the "+" symbol and add them through the window that opened.



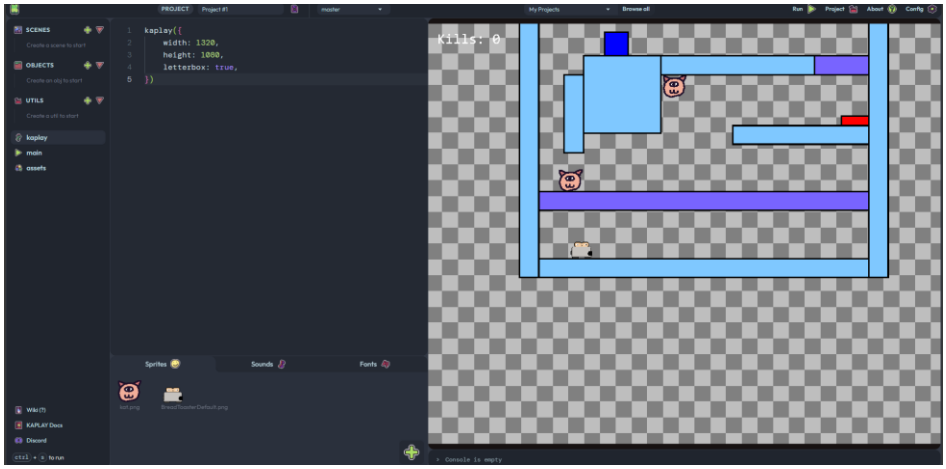
This section will then show the newly added art. (you may need to click anywhere random on the screen to make them appear)

Now, to use the art, you need to just drag and drop the art from the Sprites section to the code. Do so inside the “assets” file found at the left of the screen. Drag n Droping an image will automatically write the syntax and pathing needed for the loading of it.

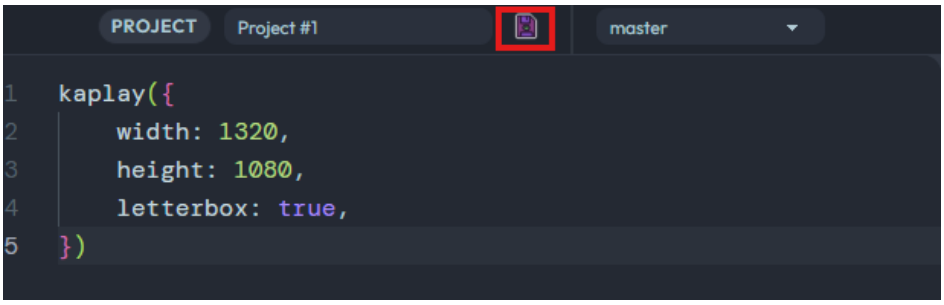
Lastly, go to the “kaplay” file on the left side of the screen and make sure it looks like this:



Finally, now by using ctrl + s or pressing the “Run” button on the top right of the screen, you will run the script and will be able to play the game with no error message.



In order to keep the project saved, click the purple floppy disk.



Toast N' guns

What is it?

Toast N' Guns is a 2D vertical side scroller, run and gun, RPG roguelike game. In this game, you play as a toast inside of a toaster with a blaster. The purpose of the game is to get to the end of the level(s) without dying in order to continue to the next one. Of course there are obstacles. In this case, other bread types as enemies, except that they are moldy.

The game itself has no story in its gameplay at its current state, but it's supposed to be that the player (the toast inside the toaster) is attempting to flee from the house that they are located in. The house is completely moldy, housing insects and different types of moldy bread. They are attempting to stop you on your tracks to become moldy like them.

Features

Layers and Levels

The levels are supposed to take place in different areas of the house. These would be separated as "Layers". Each Layer would have 3 levels. Layer 1 taking place inside a moldy food shelf in which the player is actively trying to escape. The 3rd level of each layer would then be a simple boss fight.

Upgrades and Abilities

The roguelike element comes into play here that after a certain amount of kills, the player will level up and be able to choose between 1 of 3 randomly prompted abilities. These would change or enhance the player's experience with the character or armory that is at disposal.

When the player dies, he'll have to restart the whole game back at 0.

Arsenal

There are also guns of course. The player begins with a default gun that has infinite ammo.

While playing, the player will encounter chests that are placed around the levels. These can be opened by doing enough damage to them using your gun. When opened, it will drop a weapon pickup. Getting it will give the player a random weapon from the weapon pool.

The weapons in this pool are “Submachine Gun”, “Shotgun”, “Electrical Whip”, “RPG” and “Sniper”.

These all have different stats and attributes to them, making each better at something else or playstyle.

Code and explanations

Weapon chest and assigning the weapon to the player

This is how the chests are made. They have their own details while also having an “add()” which further adds more functionality to the object.

```
const chest1 = add([
    rect(60, 60),
    pos(50, (height() - 1960)),
    outline(4),
    area(),
    health(5),
    color(BLUE),
    "Floor",
    {
        add() {
            this.on("death", () => {
                destroy(this);
                const powerup = add([
                    rect(50, 50),
                    pos(this.pos.x, this.pos.y),
                    area(),
                    outline(4),
                    color(127, 200, 255),
                    body({ isStatic: true }),
                    "gPower"
                ])
            });
```

This states that on death of the chest, it will create a new component called powerup.

This is how the player gets one of the guns randomly upon colliding with the weapon pickup (gpower).

```
kat.onCollide("gPower", (gPower) => {
    let rn = Math.floor(rand(4));
    if (rn === 0) {
        gun = "Machine Gun";
        bulletCount = 20;
    } else if (rn === 1) {
        gun = "Shotgun";
        bulletCount = 10;
    } else if (rn === 2) {
        gun = "Electrical Whip";
        bulletCount = 25;
    } else if (rn === 3) {
        gun = "Sniper";
        bulletCount = 7;
    } else if (rn === 4) {
        gun = "RPG";
        bulletCount = 10;
    }
```

The bullet count will then also be different depending on the weapon that the player receives.

```
if (showAmmo === null) {
    showAmmo = add([
        text(bulletCount),
        pos(24, 72),
        { value: 0 },
        fixed(),
    ])
} else {
    showAmmo.text = bulletCount;
}


destroy(gPower);
```

This is how I do to show the bullet amount on the screen. Lastly, destroy the weapon pickup.

For the submachine gun, the player is able to hold down left mouse click to continuously shoot due to the gun being in full auto.

```
onMouseDown(() => {
    if (canShoot == true) {
        if (gun == "Machine Gun") {
            if (bulletCount > 0) {
                bulletCount--;
            }

            spawnBullet();
            canShoot = false;

            wait(0.1, () => {
                canShoot = true;
            })
        }
    }
})
```

In contrast, all other weapons are semi auto, which makes the player have to press the left mouse click every time they want to shoot.

```
onMousePress(() => {
    if (gun == "default") {
        spawnBullet();
    } else if (gun == "Shotgun") {
        if (canShoot == true) {
            if (bulletCount > 0) {
                bulletCount--;
            }

            spawnBullet();
            canShoot = false;

            wait(0.8, () => {
                canShoot = true;
            })
        }

    } else if (gun == "Electrical Whip") {
        if (canShoot == true) {
            if (bulletCount > 0) {
                bulletCount--;
            }
```

Bullets

This is how the bullets are made. Depending on if the player is facing the left or right, the bullet will come out from the position of the character's cannon and travel towards that plane.

Each weapon has its own bullet size, travel time and damage. The way the bullets' damage are determined is using the tag that I've put on them. ("bullet", "shotgunBullet", "3dmg" and "rpg").

```
function spawnBullet() {

    if (gun == "default") {
        if (looking === "right") {
            const bullet = game.add([
                circle(5),
                pos(kat.pos.x + 42, kat.pos.y + 45),
                area(),
                move(0, 0),
                "bullet"
            ])

            bullet.onUpdate(() => {
                bullet.moveBy(15, 0)
            })



        } else if (looking === "left") {
            const bullet = game.add([
                circle(5),
                pos(kat.pos.x + 10, kat.pos.y + 45),
                area(),
                move(180, 1200),
                "bullet"
```

Level hazards

Spikes

Spikes are spread around the levels and act as a way of damaging the player.

```
const spikes = add([
    rect(70, 25),
    pos(830, height() - 395),
    outline(4),
    area(),
    body({ isStatic: true }),
    color(255, 0, 0),
    "spikes"
]);
```

This is the code to damage the player when in contact with the spikes.

```
kat.onCollideUpdate("spikes", () => {
    if (spikeable == true) {
        kat.hp--;
        debug.log(kat.hp);
        spikeable = false;

        wait(1, () => {
            spikeable = true;
        })
    }
});
```

Player

This is how the player is made. States allow us to determine what the player is doing at the moment. While I also made the code for the player getting damaged and dying inside the creation of it.

```
const kat = game.add([
    sprite("BreadToasterDefault"),
    pos(100, 1000),
    area(),
    body(),
    health(maxHP),
    "Player",
    state("grounded", ["grounded", "jump", "spinjump"]),
    {
        add() {
            this.on("hurt", () => {
                tween(WHITE, BLUE, 0.15, (p) => this.color = p);
            });

            this.on("death", () => {
                destroy(this);
                map1();
            })

            this.onCollide("Monster", () => {
                kat.hp--;
            })
        }
    }
]);
```

The way that states work is that the first state outside of the array is the default state in which the player is in all the time.

```
kat.onStateUpdate("spinjump", () => {
    const monsters = get("Monster", { recursive: true }).filter(m => kat.isColliding(m));
    monsters.forEach(m => m.hp -= 1);
    if (monsters.length > 0) {
        kat.jump();
    }
});

kat.onStateEnter("jump", () => {
    kat.jump();
});

kat.onStateUpdate("jump", () => {
    kat.enterState("grounded");
});
```

This script takes track of the state that the player is in. A state for jumping, spinjump and grounded. Spin jump is what happens when the player uses his double jump. It's a spin jump in which the player can land on enemies safely and deal damage to them with it.

```
onKeyPress("space", () => {
    if (kat.isGrounded()) {
        kat.enterState("jump");
    } else {
        if (AirJumpCount == 1) {
            kat.enterState("jump");
            kat.enterState("spinjump");
            AirJumpCount = 0;
        }
    }

});
```

Each hit will also make the player bounce off the enemy so that you can continuously bounce on enemies.

```
kat.onCollideUpdate("Wall", () => {
    if (!kat.isGrounded()) {
        AirJumpCount = 1;
    }
});

kat.onCollideUpdate("Floor", () => {
    if (!kat.isGrounded()) {
        AirJumpCount = 1;
    }
});
```

Additionally, if the player is next to a wall or a platform, they are able to do a wall jump. These don't waste your double jump or rather, "spin jump".

The game also constantly keeps track of if the player is grounded and if they have no bullets, to give them the default weapon.

```
onUpdate(() => {
    if (kat.isGrounded()) {
        kat.enterState("grounded");
        AirJumpCount = 1;
    }

    if (bulletCount == 0) {
        gun = "default";
        bulletCount = Infinity;
    }
```

Enemies

Regular enemy

The regular enemy simply walks around in a patrolling motion. Meaning that by utilizing waypoints, he'll be walking on a fixed path.

Additionally, on enemies, I have made so that when they die, they have a 1/11 chance of dropping a health pickup. This can be grabbed by the player if they are able to heal themselves. If they are at max health, they can destroy it to gain 1 additional kill score.

```
const kat2 = game.add([
    sprite("kat"),
    pos(rand(700), 822),
    area(),
    body(),
    health(3),
    state("patrol", ["patrol", "aware"]),
    patrol({
        waypoints: [
            vec2(800, 822),
            vec2(50, 822)
        ],
    }),
    "Monster",
    {
        add() {
            this.on("death", () => {
                destroy(this);
                let rnHealth = Math.floor(rand(10));
                debug.log(rnHealth);
                if (rnHealth == 0) {
                    const HealthPickup = add([
                        rect(50, 50),
                        pos(this.pos.x, (this.pos.y + 8)),
                        area(),
                        outline(4),
                        color(255, 0, 0),
                        "healthPickup",
                    ]);
                }
            });
```

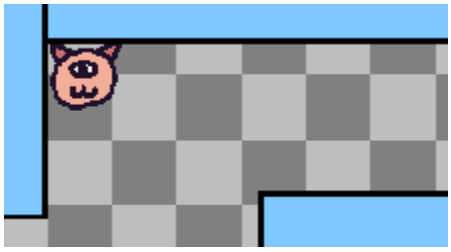
```
            this.on("hurt", () => {
                debug.log("oof");
                tween(RED, WHITE, 0.15, (p) => kat2.color = p);
            });
            this.onCollide("bullet", () => {
                this.hp--;
            });
            this.onCollide("3dmg", () => {
                this.hp = this.hp - 3;
                debug.log(this.hp);
            });
            this.onCollide("rpg", () => {
                this.hp = this.hp - 5;
            });
            this.onCollideUpdate("shotgunBullet", () => {
                this.hp--;
            });

        }
    }
]);
```

The logic of the damage of the weapons by their tag is also determined through here.

Flying type enemy

The flyer works in a similar way. However, it does not have a patrolling mechanism. Instead, they are in a resting type of state on the ceiling.



When the player gets to the floor in which the flyer is located in, they will start to chase the player, ignoring even collisions.

This is done by making sure the flyer keeps track of the player's position in the game at all time.

```
flyer1.onUpdate(() => {
    if (kat.pos.y < flyer1.pos.y + 240) {
        if (kat.pos.y > -1182) {
            flyer1.moveTo(kat.pos, 125);
        };
    };
});
```

Pause menu

The player is able to pause the game momentarily by pressing the “P” key.

```
const pauseMenu = add([
    rect(1000, 800),
    color(255, 255, 255),
    outline(4),
    anchor("center"),
    pos(getCamPos()),
    fixed(),
    layer("foreground"),
]);

pauseMenu.hidden = true;
pauseMenu.paused = true;
```

```
onKeyPress("p", () => {
    game.paused = !game.paused;
    if (game.paused) {
        pauseMenu.hidden = false;
        pauseMenu.paused = false;
    }
    else {
        pauseMenu.hidden = true;
        pauseMenu.paused = true;
    }
})
```

When leveling up, the game will also pause and open a menu in which the player would be able to choose an ability/upgrade. *Would*

```
showKills.text = "Kills: " + killCount;

if (killCount === 1 && lvl === 0) {
    killCount = 0;
    lvl = 1;
    debug.log("Level up! Level " + lvl);

    game.paused = !game.paused;
    if (game.paused) {
        pauseMenu.hidden = false;
        pauseMenu.paused = false;
    }
    else {
        pauseMenu.hidden = true;
        pauseMenu.paused = true;
    }
}
```

Camera

Very simple, the camera is set in a way to have the level centered and the player centered.

```
setCamPos(425, kat.pos.y);
```

Gravity

This gives the game gravity for the physics to work properly.

```
setGravity(1600);
```

Level layout

Normal floor

All the floors that are colored in cyan are just basic floors for the player to walk on.

```
const floor = add([
    rect(850, 48),
    pos(50, height() - 28),
    outline(4),
    area(),
    body({ isStatic: true }),
    color(127, 200, 255),
    "Floor"
]);
```

Then there are lightFloors. These have “PlatfromEffector()” which allow the player to jump through them, but not go back down unless codded in such a way. These are colored purple-ish blue.

```
const lightFloor = add([
    rect(850, 48),
    pos(50, height() - 200),
    outline(4),
    area(),
    body({ isStatic: true }),
    color(120, 100, 255),
    platformEffector(),
    "lightFloor"
]);
```