Platformer Feature Summary

01_Movement

This is the initial commit of the project. It sets up the basic file structure and includes the initial game assets. The player character can move left and right, and jump. There is no gravity, so the player will continue to move in the direction of the last keypress.

02_Player_Sprites

This tag adds sprites for the player character. The player character now has animations for idle, run, jump, and fall. The code is updated to switch between these sprites based on the player's actions.

03_Comments

This tag adds comments to the code to make it more readable and easier to understand.

04_Jumping_Xtras

This tag adds extra features to the jumping mechanic. The player can now double jump, and the jump height is variable based on how long the jump button is held down.

05_Obstacles

This tag adds obstacles to the game. The player can now collide with boxes, which will impede their movement.

06_Gameplay_Tweaks

This tag includes a number of gameplay tweaks to improve the feel of the game. The player's movement is now more responsive, and the collision detection has been improved.

07_Pickups_Sound_Score

This tag adds pickups, sound effects, and a scoring system to the game. The player can now collect cherries to increase their score, and there are sound effects for picking up cherries and for dying.

08_Globals_Fonts_DrawingText

This tag adds global variables, fonts, and the ability to draw text to the screen. The player's score is now displayed on the screen, and there is a "GOD MODE" text that appears when god mode is enabled.

09_Debug_TerminalV

This tag adds a debug terminal that displays the player's horizontal and vertical speed. This is useful for debugging the game's physics.

10_Spikes_Die_Transition

This tag adds spikes to the game, which will kill the player on contact. When the player dies, there is a fade-to-black transition before the level restarts.

11_Full_Room_Transition_Functions

This tag adds functions for transitioning between rooms. The player can now move between rooms, and the game will remember the player's position and score.

12_God_Mode

This tag adds a "god mode" to the game. When god mode is enabled, the player is invincible and cannot be killed by spikes.

13_Soundtrack_MgrIcons

This tag adds a soundtrack to the game, as well as icons for the game manager, music manager, and spawn marker.

01 – Basic Player Movement

This is the very first commit of the game, setting up the foundation for everything that comes after. Think of it as building the basic skeleton of our game. It introduces the player character, the solid ground for the player to stand on, and the initial code to make the player move and jump.

Key Files Created

The most important files for a beginner to understand in this commit are the ones that control the player. In GameMaker, objects have different "events" that hold code. The two we're focusing on are:

- `objects/oPlayer/Create_0.gml`: This code runs only **once** when the player object is first created in the game room. It's used to set up all the starting variables.
- `objects/oPlayer/Step_0.gml`: This is the player's "brain." The code here runs over and over again, once for every single frame of the game. This is where we handle continuous actions like checking for key presses, moving, and applying gravity.

Code Breakdown for Beginners

Let's look at what the code in these two files does, line by line.

`Create_0.gml` (Setting Up the Player)

This file initializes the variables that will control our player's physics.

```
hsp = 0;
vsp = 0;
dir = 0;
walk_speed = 3;
grv = 0.3;
jump_speed = -8;
on_ground = false;
```

- `hsp` and `vsp`: These stand for "horizontal speed" and "vertical speed." They start at 0 because the player isn't moving when they first appear.
- `walk_speed = 3`: This determines how fast the player moves left or right. A higher number means a faster player.
- `grv = 0.3`: This is our "gravity." In each frame, this value will be added to the vertical speed to pull the player down.
- `jump_speed = -8`: This is how powerful the player's jump is. In GameMaker, the top of the screen is Y-coordinate 0, and the bottom is a higher number. So, to move up, we need a *negative* vertical speed.
- `on_ground = false`: This is a simple true/false flag that we'll use to check if the player is currently standing on the ground. This is important so we can prevent the player from jumping in mid-air.

`Step_0.gml` **(Making the Player Move)**

This file does all the heavy lifting every frame.

1. **Check for Player Input:**

```
var dir = keyboard_check(vk_right) - keyboard_check(vk_left);
var jump = keyboard_check_pressed(vk_space);
```

   - The first line is a clever trick. `keyboard_check` returns `1` if the key is pressed and `0` if it's not. So, if the right arrow is pressed, `dir` becomes `1 - 0 = 1` (move right). If the left arrow is pressed, `dir` becomes `0 - 1 = -1` (move left). If neither or both are pressed, it's `0`.
   - The second line checks if the spacebar was just pressed in this frame.

2. **Calculate Speed and Apply Gravity:**

```
hsp = walk_speed * dir;
vsp += grv;
```

   - The horizontal speed is simply our `walk_speed` multiplied by the direction (`-1`, `0`, or `1`).
   - Gravity is added to the vertical speed, constantly pulling the player down.

3. **Jumping Logic:**

```
if(jump && on_ground) {
    vsp += jump_speed;
    on_ground = false;
}
```

   - If the jump key was pressed AND the player is on the ground, we give the player a burst of upward speed (our negative `jump_speed`) and set `on_ground` to `false`.

4. **Collision Detection (The Smart Part):** Before actually moving the player, the code checks if a move would cause a collision. This prevents the player from getting stuck in walls.

   - **Horizontal Collision:**

```
if(place_meeting(x+hsp, y, oBlock)) {
    while (!place_meeting(x + sign(hsp), y, oBlock)) {
        x += sign(hsp);
    }
    hsp = 0;
}
```

This block says: "If I'm *about to* hit a solid block horizontally, don't make the full move. Instead, move me pixel by pixel until I'm right next to the block, and then stop my horizontal speed."

- **Vertical Collision:**

  ```
  if(place_meeting(x, y+vsp, oBlock)){
      while(!place_meeting(x, y + sign(vsp), oBlock)) {
          y += sign(vsp);
      }
      vsp = 0 on_ground
      = true;
  }
  ```

  This does the same for vertical movement. If the player is about to hit the floor, it moves them right up to it, stops their vertical speed, and sets `on_ground` to `true` so they can jump again.

5. **Final Movement:**

   ```
   x += hsp;
   y += vsp;
   ```

   After all the checks and calculations, the player's final `x` and `y` coordinates are updated for this frame.

Other Files

Many other files were added, including all the game's art assets and configuration files. For a beginner, the most important thing to know is that these files set up the project's structure, define what a "solid block" looks like (`sBlock` and `oBlock`), and create the initial game room (`Room1`) where the player and blocks are placed.

02 - Player Sprite Animation

This commit introduces animations for the player character. Here's a breakdown of the changes for a beginner:

Summary

The main goal of this update was to make the player character look more alive. Instead of being a static image, the player now has different animations depending on what they are doing: standing still (idle), running, jumping, and falling.

Code Changes (`objects/oPlayer/Step_0.gml`)

This is the file that controls the player's logic every frame. The following was added:

1. **Flipping the Sprite:**

   ```
   if (dir!=0){
       image_xscale = dir;
   }
   ```

   This code checks if the player is moving left (`dir` = -1) or right (`dir` = 1). If they are, it flips the player's sprite horizontally to match the direction of movement. `image_xscale` is a built-in variable that controls the horizontal scale of the sprite. Setting it to -1 flips it.

2. **Switching Animations:** A new block of `if/else if` statements was added to change the player's sprite based on their current action.

   - `if(on_ground && hsp == 0)`: If the player is on the ground and not moving horizontally (`hsp` is horizontal speed), it sets the sprite to `sPlayer_idle`.
   - `else if(!on_ground && vsp < 0)`: If the player is in the air and moving upwards (`vsp` is vertical speed), it sets the sprite to `sPlayer_jump`.
   - `else if(!on_ground && vsp > 0)`: If the player is in the air and moving downwards, it sets the sprite to `sPlayer_fall`.
   - `else if(on_ground)`: If the player is on the ground and moving (the previous `if` for being idle was false), it sets the sprite to `sPlayer_run`.

New Assets

To make the animations work, new sprites were created:

- `sPlayer_idle`: An animation for when the player is standing still.
- `sPlayer_run`: An animation for when the player is running.
- `sPlayer_jump`: A sprite for when the player is jumping (moving up).
- `sPlayer_fall`: A sprite for when the player is falling (moving down).

In simple terms, this commit gives the player character visual life by adding animations that change based on the player's actions, making the game more dynamic and engaging.03 - Adding Explanations to Code

This commit is a small but very important one for making our game easier to understand and work on in the future. No new features were added. Instead, the focus was entirely on **commenting the code**.

What Are Comments?

In programming, comments are lines of text that the computer completely ignores. They are written purely for humans to read. You can spot them in GameMaker Language (GML) because they start with `//`.

Good comments explain the *why* behind a piece of code, not just the *what*. They act like little notes to yourself or to other programmers to make the logic clearer.

Code Changes (`objects/oPlayer/Step_0.gml`)

The only file changed in this commit was the player's `Step` event, which contains all the logic for movement and animation. Let's look at the new comments that were added.

```
// Get keyboard input
var dir = keyboard_check(vk_right) - keyboard_check(vk_left); var
   jump = keyboard_check_pressed(vk_space);

// Set horizontal and vertical speed increments for this frame hsp
   = walk_speed * dir;
vsp += grv;

// Are we jumping? if(jump
   && on_ground) {
vsp += jump_speed;
       on_ground = false;
   }

// Set the correct sprite if(on_ground
   && hsp == 0){
// ... code to set idle sprite ...
   }
// ... other sprite logic ...

// Update horizontal movement based on impending collision if
   (dir!=0){
image_xscale = dir;
   }
if(place_meeting(x+hsp, y, oBlock)) {
// ... collision code ...
   }

// Update vertical movement based on impending collision
   if(place_meeting(x, y+vsp, oBlock)){
// ... collision code ...
   }
```

```
// Finally, set the coordinate (x,y) position of the sprite for this frame
x += hsp;
y += vsp;
```

As you can see, the code itself didn't change. The new `//` lines simply break the `Step` event down into logical sections:

1. Getting player input.
2. Calculating speed and gravity.
3. Handling the jump.
4. Choosing the right animation.
5. Dealing with collisions.
6. Applying the final movement.

This makes the code much more readable. Now, if you come back to this file weeks later, you can quickly understand the purpose of each block of code without having to re-read and decipher every single line. This is a fundamental practice in good coding!

04 – Advanced Jumping Mechanics

This commit makes the player's jump feel much more dynamic and skillful. Instead of a single, fixed jump, two new features were added: **Double Jumping** and **Variable Jump Height**. This gives the player more control in the air.

Code Changes

The changes were made in the player's `Create` and `Step` events.

`Create_0.gml` (Setting Up New Jump Variables)

New variables were added to keep track of the new jump states.

```
grv = 0.4;
jump_speed = -5;
jump_timer = 0;
jump_hold_max = 10;
jump_count = 0;
max_jumps = 2;
```

- `grv` and `jump_speed` were tweaked slightly to feel better with the new mechanics.
- `jump_timer` and `jump_hold_max`: These control the **variable jump height**. The player can hold the jump button for up to 10 frames to get a little extra boost.
- `jump_count` and `max_jumps`: These control the **double jump**. The player is allowed a maximum of 2 jumps before they have to touch the ground again.

`Step_0.gml` (Implementing the New Logic)

The core logic in the `Step` event was updated to handle these new mechanics.

1. **Reading More Input:**

   ```
   var jump_pressed = keyboard_check_pressed(vk_space);
   var jump_held_now = keyboard_check(vk_space);
   ```

   - The code now checks for two things: if the jump button was *just pressed* (`jump_pressed`) and if it's *currently being held down* (`jump_held_now`).

2. **The New Jump Logic:**

   ```
   // Double Jump Logic
   if (jump_pressed && jump_count < max_jumps) {
       vsp = jump_speed;
       jump_timer = jump_hold_max;
       on_ground = false;
   ```

```
        jump_count++;
    }

    // Variable Jump Height Logic
    if (!on_ground && jump_held_now && jump_timer > 0) {
        vsp -= 0.5; // extra upward force
        jump_timer -= 1;
    }
```

- The first `if` statement now allows a jump as long as the `jump_count` is less than `max_jumps` (2). Each time the player jumps, `jump_count` goes up by one.
- The second `if` statement gives the player a small upward boost (`vsp -= 0.5`) for every frame they hold down the jump button, but only for a maximum of 10 frames (`jump_timer`).

3. **Resetting the Jumps:**

```
if(place_meeting(x, y+vsp, oBlock)){
    // ... collision code ...
    on_ground = true;
    jump_count = 0; // Reset jump counter
}
```

- Crucially, whenever the player lands on the ground, the `jump_count` is reset back to `0`, allowing them to double jump all over again.

Summary for Beginners

Think of this update as giving the player more tools for navigating the world.

- **Double Jump:** You can now press the jump button a second time while in the air to get another boost. This is essential for clearing larger gaps.
- **Variable Jump:** Tapping the jump button gives you a small hop, while holding it down gives you a higher, more powerful jump.

These two features are staples of the platformer genre and make controlling the character feel much more fluid and satisfying.

05 – Adding Solid Boxes

This commit introduces a new type of object to the game world: the **Box**. While it might seem simple, this is a fundamental step in making the level more interactive and challenging. The player can stand on boxes and must navigate around them, just like the solid ground blocks.

Key Changes

1. **New Object (`oBox`)**: A new object, `oBox`, was created. It uses the `sBox` sprite for its appearance.
2. **Making Boxes Solid**: The player's code was updated to treat these new boxes as solid objects that can be collided with.

Code Breakdown for Beginners

The changes are straightforward and show how to handle different types of solid objects efficiently.

`Create_0.gml` (Defining What's Solid)

A new variable was added to the player's `Create` event to keep track of all the objects that should be treated as solid.

```
solid_objects = [oBlock, oBox];
```

- `solid_objects`: This is an **array**, which is just a list of items. In this case, we're making a list of all the objects the player should not be able to pass through. By putting `oBlock` and the new `oBox` in this list, we can check for collisions against both at the same time.

`Step_0.gml` (Checking Collisions with All Solid Objects)

The collision code in the player's `Step` event was updated to use our new `solid_objects` array.

Old Horizontal Collision Code:

```
if(place_meeting(x+hsp, y, oBlock)) { ... }
```

New Horizontal Collision Code:

```
if(place_meeting(x+hsp, y, [oBlock, oBox])) { ... }
// Or even better, using the array:
if(place_meeting(x+hsp, y, solid_objects)) { ... }
```

- Instead of only checking for collisions with `oBlock`, the code now checks for collisions with *any* of the objects in our `solid_objects` list. The same change was made for the vertical collision check.

This is a great programming practice. If we want to add more solid objects later (like pipes, moving platforms, etc.), we just have to add them to the `solid_objects` list in the `Create` event, and all the collision code will work automatically without needing any other changes.

New Assets

- `sBox` **(Sprite)**: The image for the wooden crate.
- `oBox` **(Object)**: The actual object placed in the room that uses the `sBox` sprite.

Summary for Beginners

This update adds a new obstacle, the **Box**, to the game. The key takeaway is how the code was organized to handle multiple types of solid objects cleanly. By using an array (`solid_objects`), we've made the collision system expandable and easy to manage, which is a crucial skill for building more complex games.

06 - Refining Player Movement

This commit is all about "game feel." The player could move and jump before, but this update makes that movement feel smoother, more responsive, and more like a professional platformer game. This is achieved by adjusting physics variables and adding **air control**.

Code Changes

The changes focus on the player's `Create` and `Step` events.

`Create_0.gml` (New Physics Variables)

Several variables were tweaked or added to fine-tune the player's movement.

```
walk_speed = 2.5;
jump_speed = -4;
air_accel = 0.5;
air_max = walk_speed;
```

- `walk_speed` and `jump_speed` were slightly reduced. This is a common part of game development called "balancing"—tweaking numbers until the movement feels just right.
- `air_accel = 0.5`: This is the **air acceleration**. It controls how quickly the player can change direction while in mid-air.
- `air_max = walk_speed`: This sets a speed limit for how fast the player can move horizontally while in the air, preventing them from gaining infinite speed.

`Step_0.gml` (Implementing Air Control)

The biggest change is in how horizontal speed (`hsp`) is calculated.

Old Code:

```
hsp = walk_speed * dir;
```

This line meant the player had instant, full-speed movement, which can feel a bit robotic, especially in the air.

New Code:

```
if (on_ground) {
    hsp = walk_speed * dir; // Normal grounded movement
} else {
    // In air: preserve momentum, but allow some control
    hsp += dir * air_accel;
    hsp = clamp(hsp, -air_max, air_max);
}
```

This new logic creates a major difference in game feel:

- **On the Ground**: Movement is still instant and responsive.
- **In the Air**: This is the key change.
  - `hsp += dir * air_accel;`: Instead of instantly setting the speed, the code now *adds* a small amount of speed in the direction the player is pressing. This creates a feeling of momentum and weight.
  - `hsp = clamp(hsp, -air_max, air_max);`: This line uses the `clamp` function to make sure the horizontal speed in the air never goes above the `air_max` limit we set earlier.

A small tweak was also made to the variable jump height to feel better with the new gravity:

```
// Old line: vsp -= 0.5;
vsp -= 0.4; // New line: extra upward force
```

Summary for Beginners

This commit makes the player's movement, especially in the air, feel much less stiff and more natural.

- **Before**: When you jumped, you were locked into that jump's path. Changing direction in the air was instant and jerky.
- **After**: Now, you have **air control**. You can gently influence your movement left and right while falling, allowing for more precise landings and a feeling of momentum. This is a subtle but hugely important feature for making a platformer fun to play.

07 - Adding Collectibles and Feedback

This commit adds some of the most classic and important elements of a platformer game: things to collect, sound effects to make actions feel rewarding, and a score to track the player's progress.

Key Features

1. **Cherries (`oCherry`)**: A new collectible object is introduced.
2. **Scoring**: The player now has a score that increases when they collect a cherry.
3. **Sound Effects**: A sound now plays when a cherry is collected.

Code Breakdown for Beginners

`oPlayer` Object Changes

1. **`Create_0.gml` (Adding a Score Variable)**

```
player_score = 0;
```

   - A new variable, `player_score`, is added to the player object. It's initialized to `0` when the game starts.

2. **`Collision with oCherry` Event (The Core Logic)** A brand new event was added to the player object: the **Collision Event** with `oCherry`. This code runs automatically whenever the player touches a cherry.

```
// Destroy the cherry
with (other) {
    instance_destroy();
    audio_play_sound(sndCherryPickup, 10, false);
}

// Update the Player's score
player_score += 100;
show_debug_message("Player Score: " + string(player_score));
```

   - `with (other)`: This is a special block that makes the code inside it apply to the *other* object in the collision (in this case, the cherry).
   - `instance_destroy()`: This function destroys the cherry instance, making it disappear from the game.
   - `audio_play_sound(...)`: This plays the new cherry pickup sound effect.
   - `player_score += 100`: This adds 100 points to the player's score.
   - `sh0w_debug_message(...)`: This is a temporary line for developers. It prints the player's current score to the output log so we can make sure it's working correctly before we display it on the screen.

New Assets

- `oCherry` **(Object) and** `sCherry` **(Sprite)**: The new collectible cherry object and its animated sprite.
- `sndCherryPickup` **(Sound)**: The sound effect that plays when a cherry is collected.

Summary for Beginners

This commit makes the game world more interactive and gives the player a clear goal.

- You can now **collect cherries** scattered throughout the level.
- Collecting a cherry makes it disappear, plays a satisfying **sound effect**, and **increases your score**.

This is a classic feedback loop in game design: the player performs an action (collecting), and the game responds with positive feedback (sound, score increase), which makes the action feel rewarding and encourages the player to do it again.

08 - Displaying the Score

This commit builds directly on the last one. We had a scoring system, but the player couldn't see their score! This update focuses on creating a **Heads-Up Display (HUD)** to show the score on the screen. It also introduces a more advanced way to handle variables: **global variables**.

Key Features

1. **Global Variables**: The score is moved from a regular variable to a `global` one, making it accessible from anywhere in the game.
2. **Custom Font**: A new pixel-art font is added to the game for displaying the score in a style that matches the graphics.
3. **Drawing to the Screen**: Code is added to a manager object to draw the score text to the screen every frame.

Code Breakdown for Beginners

`scripts/globals.gml` (A Central Place for Important Variables)

This is a new and very important type of asset: a **Script**. In GameMaker, scripts are used to store functions or, in this case, to declare variables that need to be accessed from anywhere in the game.

```
// GLOBAL VARIABLES & CONSTANTS
#macro PICKUP_CHERRY  100
#macro SCORE_NUM_ZEROS6

////////////////////////////
global.score = 0;
```

- `#macro`: This is a way to create a **constant**. Think of it as giving a nickname to a value. Now, instead of typing `100` every time we want to give points for a cherry, we can just type `PICKUP_CHERRY`. If we later decide cherries should be worth 150 points, we only have to change it in this one place!
- `global.score = 0`: This creates a **global variable**. By putting `global.` in front of `score`, we are making it accessible from any object at any time. This is much better than storing the score on the player object, especially as the game gets more complex.

`oPlayer` Object Changes

The player's code was updated to use the new global score.

- `Create_0.gml`: The line `player_score = 0;` was **removed**. The score is now handled by the `globals` script.
- `Collision_oCherry.gml`: The line `player_score += 100;` was changed to `global.score += PICKUP_CHerry;`.

`oGameManager` Object Changes

This object is now responsible for drawing the HUD.

- `Draw GUI` **Event (`Draw_64.gml`)**: A new event was added to the Game Manager. The `Draw GUI` event is special because it draws things on a layer that is separate from the game world. This means the score will stay fixed in the top-left corner of the screen, even when the camera follows the player.

```
draw_set_font(fnt_score);
draw_set_color(make_colour_rgb(255, 75, 172));
draw_set_halign(fa_left);
draw_set_valign(fa_top);
draw_text(40, 5, "SCORE: " + string_repeat("0", 6 -
string_length(string(global.score))) + string(global.score));
```

1. It sets the font to our new pixel art font (`fnt_score`).
2. It sets the draw color to a nice pink.
3. It aligns the text to the top-left.
4. The `draw_text` line is a bit complex, but it does something cool: it draws the word "SCORE: " followed by the player's score, padded with leading zeros (e.g., "000100" instead of "100").

Summary for Beginners

This commit creates the game's first piece of user interface (UI).

- The score is now managed by a **global variable**, which is a best practice for data that needs to be accessed everywhere.
- A custom **font** was added to make the text match the game's art style.
- The **score is now visible on the screen** during gameplay, giving the player constant feedback on their progress.

09 - Fine-Tuning Physics

This commit introduces a physics concept called **Terminal Velocity** and adds some helpful debug messages to let the developer "see" the player's speed.

Key Features

1. **Terminal Velocity**: The player can no longer accelerate downwards infinitely. Their falling speed is now capped at a maximum value.
2. **Debug Messages**: The player's horizontal and vertical speeds are now printed to the output console, which is a useful tool for debugging and balancing the game's physics.

Code Breakdown for Beginners

`Create_0.gml` (Adding the Terminal Velocity Variable)

A new variable was added to the player's `Create` event to define the maximum falling speed.

```
terminal_vsp = 8;
```

- `terminal_vsp`: This variable holds the maximum speed the player can reach while falling. Without this, gravity would make the player fall faster and faster forever, which can lead to unpredictable behavior and bugs.

`Step_0.gml` (Implementing Terminal Velocity and Debugging)

1. **Capping the Fall Speed:**

   ```
   // Make sure vsp never exceeds terminal velocity
   vsp = min(vsp, terminal_vsp);
   ```

   - This line is the core of the new feature. The `min()` function takes two numbers and returns the smaller one.
   - If the player's vertical speed (`vsp`) is less than `terminal_vsp` (e.g., 5), it stays as it is.
   - If gravity accelerates the `vsp` to be greater than `terminal_vsp` (e.g., 9), this line will force it back down to 8. This ensures the player never falls faster than the maximum speed we set.

2. **Adding Debug Messages:**

   ```
   // Debug messages for speed
   if hsp != 0 show_debug_message("hsp: " + string(hsp));
   if vsp != 0 show_debug_message("vsp: " + string(vsp));
   ```

  - These lines are purely for the developer. They check if the player is moving horizontally or vertically, and if so, they print the current speed values to the output log in GameMaker. This is an invaluable tool for understanding exactly what the code is doing and for fine-tuning variables like `walk_speed`, `grv`, and `jump_speed` to get the game feeling just right.

Summary for Beginners

This commit is about polishing the game's physics and adding tools for the developer.

- **Terminal Velocity** makes the player's fall more controlled and realistic, preventing them from reaching ridiculously high speeds.
- The **Debug Messages** are like a car's speedometer for the developer, showing the exact numbers behind the player's movement so they can make informed decisions when balancing the game.

10 - Adding Hazards and a Death Sequence

This commit introduces a major gameplay element: **danger**. Spikes are added to the level, and for the first time, the player can "die." This commit also creates a smooth screen-fade transition for when the player dies and the level restarts.

Key Features

1. **Spikes (`oSpike`)**: A new hazardous object that will kill the player on contact.
2. **Death State**: The player now has a "dead" state to prevent them from moving after being hit.
3. **Death Animation and Sound**: When the player dies, they play a "hit" animation and a death sound effect.
4. **Fade-to-Black Transition**: A new controller object (`oFadeController`) is created to smoothly fade the screen to black before restarting the room.

Code Breakdown for Beginners

`oPlayer` Object Changes

1. `Create_0.gml` **(New "isDead" Flag)**

   ```
   isDead = false;
   ```

   - A new variable, `isDead`, is added. This is a simple true/false flag that we'll use to track if the player is currently alive or dead.

2. `Step_0.gml` **(Stopping Player Movement on Death)**

   ```
   if (isDead) exit;
   ```

   - This line is added right at the beginning of the `Step` event. `exit` immediately stops running any more code in the event. This is a simple and effective way to freeze the player in place as soon as they die.

3. `Collision with oSpike` **Event (The Death Logic)** This is the core of the new feature. This code runs when the player touches a spike.

   ```
   if (!isDead) {
       isDead = true;
       sprite_index = sPlayer_hit;
       image_index = 0;
       image_speed = 0;
       audio_play_sound(sndDie, 10, false);
   ```

```
        alarm[0] = 60;
    }
```

- `if (!isDead)`: This check is important to make sure the death sequence only runs once.
- `isDead = true;`: The player is now officially dead.
- `sprite_index = sPlayer_hit;`: The player's sprite is changed to the new "hit" animation. o `image_index = 0; image_speed = 0;`: These lines freeze the animation on its first frame. o `audio_play_sound(...)`: Plays the death sound effect.
- `alarm[0] = 60;`: This is a crucial part. It sets **Alarm 0** to run after 60 frames (which is 1 second in a 60 FPS game).

4. `Alarm 0` **Event (Triggering the Restart)** This new event runs exactly 60 frames after the player dies.

```
var f = instance_create_layer(0, 0, "Instances", oFadeController);
f.next_room = -1;
```

- This code creates our new `oFadeController` object, which will handle the screen fade. Setting `next_room` to `-1` tells the controller to restart the current room.

`oFadeController` **Object (The Transition Manager)**

This new object is dedicated to handling screen transitions.

- `Create` **Event**: It initializes variables for the fade's transparency (`fade_alpha`) and speed.
- `Step` **Event**: This is where the magic happens. It gradually increases `fade_alpha` from 0 (clear) to 1 (fully black). Once the screen is black, it restarts the room and then fades back in by decreasing the alpha. When it's fully faded in, the controller destroys itself.
- `Draw GUI` **Event**: This draws a black rectangle over the entire screen, using `fade_alpha` to control its transparency.

Summary for Beginners

This commit adds a complete "death cycle" to the game.

1. The player touches a **spike**.
2. The player enters a **dead state**, plays a hit animation and sound, and can no longer move.
3. An **alarm** is set.
4. After one second, the alarm triggers a **fade-to-black** transition.
5. Once the screen is black, the **room restarts**, putting the player back at the beginning, ready to try again.

11 – Seamlessly Connecting Levels

This commit builds a robust system for moving the player between different rooms or levels. It introduces several new objects and concepts to make the transitions smooth and professional, ensuring the player appears in the correct spot in the next room.

Key Features

1. **Persistent Player and Game Manager**: The player and game manager objects now persist between rooms, meaning they don't get destroyed and re-created every time the level changes. This allows them to remember information, like the player's score.
2. **Room Exits (`oExit`)**: A new object that acts as a doorway to another room. It can be configured to lead to any room in the game.
3. **Spawn Markers (`oSpawnMarker`)**: An invisible object that tells the game where the player should appear when entering a new room.
4. **Centralized Transition Logic**: A global function, `transition_to_room()`, is created to handle all the logic for fading out and changing rooms.

Code Breakdown for Beginners

`globals.gml` (The New Transition Function)

A powerful new function was added to the global script.

```
function transition_to_room(target_room) {
    if (instance_exists(oFadeController)) return; // Don't start a new transition if one is already happening

    var f = instance_create_layer(0, 0, "Instances", oFadeController);
    f.next_room = target_room;

    global.transitioned = (target_room != -1); // Set a flag that we are changing rooms, not just restarting

    audio_stop_all(); // Stop the current room's music
}
```

- This function can be called from anywhere. You just tell it which room to go to (`target_room`). It creates the fade controller and sets a global flag so the game knows a transition is happening.

`oPlayer` Object (Making it Smarter)

1. **Persistent Property**: The `oPlayer` object (and `oGameManager`) was marked as **"Persistent"** in the GameMaker editor. This is a simple checkbox that tells the game *not* to destroy this object when a new room loads.

2. `Create_0.gml` **(New Variables and a Reset Function)**

   - `spawn_x`, `spawn_y`, `spawn_facing`: These variables are added to remember the player's starting position in a room, so when they die, they respawn there.
   - `resetPlayer()`: A function is created to reset all the player's stats (health, speed, position) back to their starting values.

3. `Room Start` **Event (**`Other_4.gml`**)** This is a new event that runs automatically whenever a room starts (or restarts).

```
if (global.transitioned) {
    // We came from another room
    resetPlayer();
    var sp = instance_find(oSpawnMarker, 0); // Find the spawn marker in the
new room
    x = sp.x;
    y = sp.y;
    // ... update spawn point for this new room ...
    global.transitioned = false; // Clear the flag
} else {
    // We died and are restarting the same room
    resetPlayer();
}
```

   - This logic checks the `global.transitioned` flag. If it's true, it knows the player has come from another level and moves them to the `oSpawnMarker`'s position. If it's false, it means the player died, so it just puts them back at the start of the current level.

`oExit` Object (The Doorway)

This new object is simple but powerful.

- **Properties**: It has variables you can set in the room editor, like `target_room` (which room it leads to) and `next_facing` (which way the player should be facing when they arrive).
- **Collision with Player**: When the player touches an exit, it calls our new `transition_to_room()` function, telling it to go to the `target_room`.

Summary for Beginners

This commit creates a complete, professional system for moving between levels.

1. The player touches an **Exit Door (**`oExit`**)**.
2. The exit calls the global `transition_to_room()` **function**.
3. This function creates the **Fade Controller**, which fades the screen to black.
4. The game loads the **new room**.
5. Because the **Player is "Persistent"**, it still exists in the new room.
6. The player's `Room Start` **event** triggers. It sees the `global.transitioned` flag is true, finds the **Spawn Marker (**`oSpawnMarker`**)** in the new level, and moves itself to that exact spot.

7. The **Fade Controller** fades the screen back in, and the player is ready to go in the new level, with their score intact.

12 - Adding a Cheat for Testing

This commit adds a classic developer tool: a "God Mode." This is a cheat that makes the player invincible, which is incredibly useful for testing levels without having to worry about dying.

Key Features

1. **God Mode Toggle**: The player can now press the "G" key to turn God Mode on and off.
2. **Invincibility**: While God Mode is active, the player will no longer die when they touch spikes.
3. **On-Screen Indicator**: Text appears on the screen to let the player know that God Mode is active.

Code Breakdown for Beginners

`globals.gml` (Tracking God Mode State)

A new global variable was added to keep track of whether God Mode is on or off.

```
global.god_mode = false;
```

- This creates a new global flag that starts as `false`. Because it's global, other objects (like the `oGameManager` that draws the text) can easily check its value.

`oPlayer` Object Changes

1. `Step_0.gml` **(Toggling God Mode)**

```
if (keyboard_check_pressed(ord("G"))) {
    global.god_mode = !global.god_mode;
    show_debug_message("### GOD MODE: " + string(global.god_mode));
}
```

   - This code is added at the very top of the player's `Step` event.
   - `keyboard_check_pressed(ord("G"))`: This checks if the "G" key was just pressed. `ord("G")` is just how the computer understands the "G" key.
   - `global.god_mode = !global.god_mode;`: This is a clever trick to flip a boolean (true/false) value. The `!` means "not." So, if `global.god_mode` is `false`, it becomes `!false` (which is `true`), and if it's `true`, it becomes `!true` (which is `false`).

2. `Collision with oSpike` **Event (Implementing Invincibility)** The code that handles the player's death was updated with one simple check.

```
if (!global.god_mode && !isDead) {
    // ... death sequence code ...
}
```

- The death sequence will now only run if `global.god_mode` is `false` AND the player isn't already dead. If God Mode is on, this entire block of code is skipped, and the player simply passes through the spikes unharmed.

`oGameManager` **Object (`Draw_64.gml`)**

The `Draw GUI` event was updated to show the player that God Mode is active.

```
if (global.god_mode) {
    draw_set_halign(fa_right);
    draw_text(room_width - 40, 10, "GOD MODE");
}
```

- This code checks the global flag every frame. If it's `true`, it draws the text "GOD MODE" in the top-right corner of the screen.

Summary for Beginners

This commit adds a powerful debugging and testing tool. By pressing "G," you can make the player invincible. This allows you to freely explore levels, test jump distances, and check level design without the frustration of repeatedly dying. The on-screen text provides clear feedback that the cheat is active.

13 - Adding Music and Improving Organization

This commit brings the game to life with background music and adds a nice organizational touch for the developers in the GameMaker IDE.

Key Features

1. **Background Music**: Each room now has its own unique soundtrack that plays on a loop.
2. **Music Manager (`oMusicManager`)**: A new dedicated object is created to handle playing the correct music for each room.
3. **IDE Icons**: The invisible "manager" objects (`oGameManager`, `oMusicManager`, `oSpawnMarker`) are given sprites. These sprites are **not visible in the game** but act as helpful icons in the room editor, making it easier for developers to see where these important objects are placed.

Code Breakdown for Beginners

`oMusicManager` Object (The DJ of the Game)

This is a new object whose only job is to play music. This is a great example of good organization—keeping different functionalities in separate, dedicated objects.

- **`Create_0.gml` (Choosing the Right Song)**

```
switch (room) {
    case Room1:
        audio_play_sound(sndRoom1_soundtrack, 1, true);
        break;
    case Room2:
        audio_play_sound(sndRoom2_soundtrack, 1, true);
        break;
}
```

  - This code runs once when the `oMusicManager` is created in a room.
  - A `switch` statement is like a series of `if` checks. It checks the value of the built-in `room` variable.
  - If the current room is `Room1`, it plays the `sndRoom1_soundtrack`. If it's `Room2`, it plays the `sndRoom2_soundtrack`.
  - `audio_play_sound(sound, priority, looping)`: The third argument, `true`, tells the sound to loop forever.

`globals.gml` (Stopping the Music)

A small but important line was added to the `transition_to_room()` function.

```
function transition_to_room(target_room) {
    // ... fade out logic ...
    // Stop soundtrack
    audio_stop_all();
}
```

- `audio_stop_all()`: This function immediately stops all sounds that are currently playing. This is crucial because when we leave a room, we want its music to stop before the new room's music begins.

Manager Icons

The `oGameManager`, `oMusicManager`, and `oSpawnMarker` objects were all assigned new sprites (`spr_GameManager`, etc.). In the object editor, the "Visible" checkbox for these objects was **unchecked**.

- **Why do this?** These objects don't need to be seen by the player, but they do need to be placed in the room by the developer. Giving them a sprite makes them appear as a clear icon in the room editor, so the developer can easily select, move, and manage them without them being invisible.

New Assets

- **`sndRoom1_soundtrack`** & **`sndRoom2_soundtrack`**: The new music files for the first two rooms.
- **`oMusicManager`**: The new object to control the music.
- **`spr_GameManager`**, **`spr_MusicManager`**, **`spr_SpawnMarker`**: The new icon sprites for the manager objects.

Summary for Beginners

This commit adds a layer of polish and atmosphere to the game.

- Each level now has its own **background music**, making the world feel more immersive.
- A dedicated **Music Manager** handles this automatically, keeping the code clean.
- Finally, adding **icons for manager objects** is a quality-of-life improvement for the developer, making the project easier to work with in the GameMaker editor.