The Pragmatic Programmers

Advanced Hands-on Rust

Level up Your Coding Skills

Herbert Wolverson

edited by Tammy Coron

CHAPTER 7

Teach Your Dragon to Fly

In this chapter, you'll learn two important aspects of 2D games: animation and physics.

Animation brings your game to life by making sprites change over time, providing visual cues as to what's happening in the game. The human eye is very sensitive to movement, and static scenes don't attract the same attention as animated scenes. Notice that with most games, almost everything is animated.

Physics also brings games to life. Players intuitively understand physics, and somewhat-accurate modeling of how the world works makes it easier for players to interact with your game.

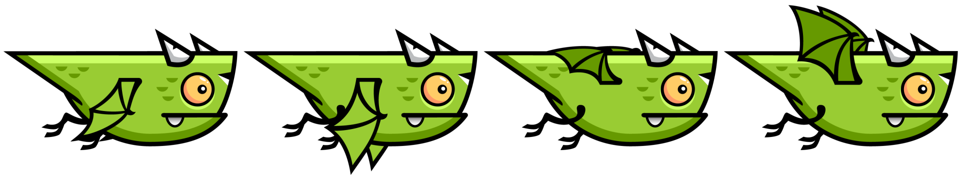
Physics and animation share a common feature: time. It shouldn't matter how fast your player's PC is running, animations need to remain smooth and consistent. More importantly, running a game on a fast PC shouldn't speed up an entity's movement within a simulation, because in a few years, as computers get faster, your "odler" game may run *too fast* to be playable.

Adding Frame-Based Animation

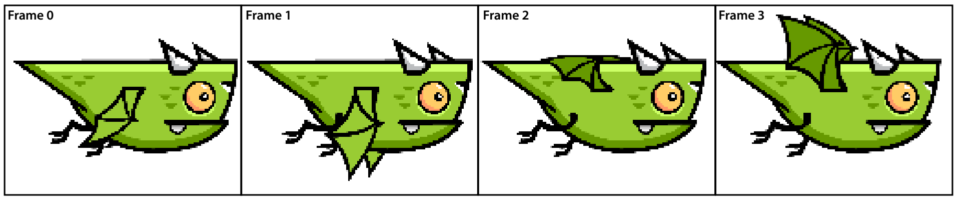
Frame-based animation works similarly to paper flip books: each page displays a picture, more precisely, a slightly different image on each page. As you flip through the book, you see the characters move. However, each page ("frame") is a static image. Unlike flip books, frame-based animation can reuse images without adding extra pages.

Frame-based animation can also be tied to instructions, such as "play this sound when this frame appears," and it's a lot easier to switch between animations on a computer than it is in a printed flip-book.

Let's get started by using the full set of animation frames for Flappy the Dragon. Bevoulin (on Open Game Art) includes four animation frames for the dragon:[1]



You can divide these images into frames, where each frame is the same size, essentially making a "digital" page in a flip book:



Looking at these frames, you don't want to simply repeat them—if you jump from frame 3 to frame 0, the dragon's wings will go from "all the way up" to "all the way down" with no in-between. Instead, the correct render order is `0, 1, 2, 3, 2, 1` (repeat). This pattern provides a smooth "flapping" cycle, with the wings working their way up, and then down again, mimicking a realistic use of wings.

Download `code/FlappyAnimation/flappy_dragon_base/flappy_sprite_sheet.png` and place the file in the `flappy_dragon/src/assets` directory. This file has been scaled to match the 1024×768 resolution of your game.

There are multiple logical layers to an animation set:

- The image itself, containing every frame. Bevy represents this as an `Image`, just like your other game sprites.
- A grid, placed over the sprite image to provide identically sized sub-images. Bevy names this your `SpriteSheet`. Each frame in the image above is a separate sprite sheet entry.
- Animations, which group animations together by name for easy access. You'll create a type for these named `PerFrameAnimation`.
- Each animation frame—containing the *index* of the sprite to use (from the `SpriteSheet`), a duration for the frame, and one or more *actions* to perform when the frame is active.

1. https://opengameart.org/content/flappy-dragon-sprite-sheets

Let's extend the asset manager to handle frame-based animations.

Defining Sprite Sheets

The first step is to add SpriteSheet handling to your asset system. You used a SpriteSheet in Playing Pig, on page ?

Open my_library/src/bevy_assets/asset_manager.rs and add a new entry to the AssetType enumeration:

FlappyAnimation/my_library/src/bevy_assets/asset_manager.rs

```
#[derive(Clone)]
pub enum AssetType {
  Image,
  Sound,
➤ SpriteSheet{tile_size: Vec2, sprites_x: usize, sprites_y: usize},
}
```

Bevy needs to know a few things about the sprite sheet, so they're included in this type:

- The size of each frame (in pixels), represented by tile_size—a Vec2 containing an x and y value.
- The number of sprites in the x dimension (columns), represented by sprites_x.
- The number of sprites in the y dimension (rows), represented by sprites_y.

Next, just like the image and audio asset types, you need to add a function to add a sprite sheet to the requested assets list. Add the new function inside the AssetManager implementation, after add_image():

FlappyAnimation/my_library/src/bevy_assets/asset_manager.rs

```
pub fn add_sprite_sheet<S: ToString>(
  mut self,
  tag: S,
  filename: S,
  sprite_width: f32,
  sprite_height: f32,
  sprites_x: usize,
  sprites_y: usize,
) -> anyhow::Result<Self> {
  let filename = filename.to_string();
  AssetManager::asset_exists(&filename)?;
  self
    .asset_list
    .push((tag.to_string(), filename, AssetType::SpriteSheet{
      tile_size: Vec2::new(
        sprite_width,
        sprite_height),
        sprites_x,
        sprites_y,
    }));
  Ok(self)
}
```

Your library consumer can now request that a sprite sheet be loaded in the same way as other assets. Next, you need to add the code to actually create a Bevy TextureAtlas.

Building Sprite Sheets

Bevy's Sprite Sheet documentation shows that creating a sprite sheet is a two-step operation. The first step loads an image, just like your previous image loading code. The second step creates a TextureAtlas structure, which refers to the image, has a Handle, and won't appear in the list of loaded assets. TextureAtlass are a *meta-asset*. Describing an interaction with physical assets depends upon the asset, but they don't actually occupy a slot in the asset manager's data store.

The difficult part with creating a meta-asset is that you have to first load the base asset—and then initialize the meta-asset, using it. The meta-asset then gains a Handle, but never has a Handle<LoadedUntypedAsset>—because no loading stage was ever invoked. This requires that the asset system load the base image—and store everything required to build the texture atlas after it has been loaded. Start by creating a new FutureAtlas type in my_library/src/bevy_assets/asset_store.rs:

FlappyAnimation/my_library/src/bevy_assets/asset_store.rs

```
pub(crate) struct FutureAtlas {
  pub(crate) tag: String,
  pub(crate) texture_tag: String,
  pub(crate) tile_size: Vec2,
  pub(crate) sprites_x: usize,
  pub(crate) sprites_y: usize,
}
```

This type will be used to store atlases that are awaiting creation. Next, extend your AssetStore type to include the list of future atlases and a HashMap to store atlases once they have been created:

FlappyAnimation/my_library/src/bevy_assets/asset_store.rs

```
  #[derive(Resource)]
  pub struct AssetStore {
    pub(crate) asset_index: HashMap<String, Handle<LoadedUntypedAsset>>,
➤   pub(crate) atlases_to_build: Vec<FutureAtlas>,
➤   pub(crate) atlases: HashMap<String, Handle<TextureAtlas>>,
  }
```

While you're in the file, add a function to retrieve texture atlases by name. Add the code inside the impl AssetStore block:

FlappyAnimation/my_library/src/bevy_assets/asset_store.rs

```
pub fn get_atlas_handle(&self, index: &str) -> Option<Handle<TextureAtlas>>
{
  if let Some(handle) = self.atlases.get(index) {
    return Some(handle.clone());
  }
  None
}
```

Now that the AssetStore supports texture atlases, you need to add some logic to create them. Open the asset_manager.rs file, Add the following code above the default option (_ =>):

FlappyAnimation/my_library/src/bevy_assets/asset_manager.rs

```
  match asset_type {
    AssetType::SpriteSheet { tile_size, sprites_x, sprites_y } => {
      // Sprite Sheets require that we load the image first, and defer
      // sheet creation to the loading menu - after the image has loaded
1     let image_handle = asset_server.load_untyped(filename);
2     let base_tag = format!("{tag}_base");
      assets
        .asset_index
3       .insert(base_tag.clone(), image_handle);

      // Now that its loaded, we store the future atlas in the asset store
4     assets.atlases_to_build.push(FutureAtlas {
        tag: tag.clone(),
        texture_tag: base_tag,
        tile_size: *tile_size,
        sprites_x: *sprites_x,
        sprites_y: *sprites_y,
      });
    }
```

1. Load the underlying image, just as you did for images previously.
2. You want to use a different tag for the underlying image. You can't have duplicate tags in a HashMap.
3. Insert the base image, identified by the new tag.
4. Add the atlas' details to the future texture atlases list you created.