2D graphics

Double buffering and sprites

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

27 października 2016

Table of contents

2D game?

Yes, this may seem strange in the contemporary world, but 2D graphics is still on the top. Why? Because

- Gamers are drawn toward 2D because of the purity and simplicity of the games.
- Developers are drawn to 2D because the typical budget and team can be much smaller.

It should be noted that many of the game topics, whether physics, sound, or UI programming, are equally applicable in both 2D and 3D games.

Not only 2D rendering foundations

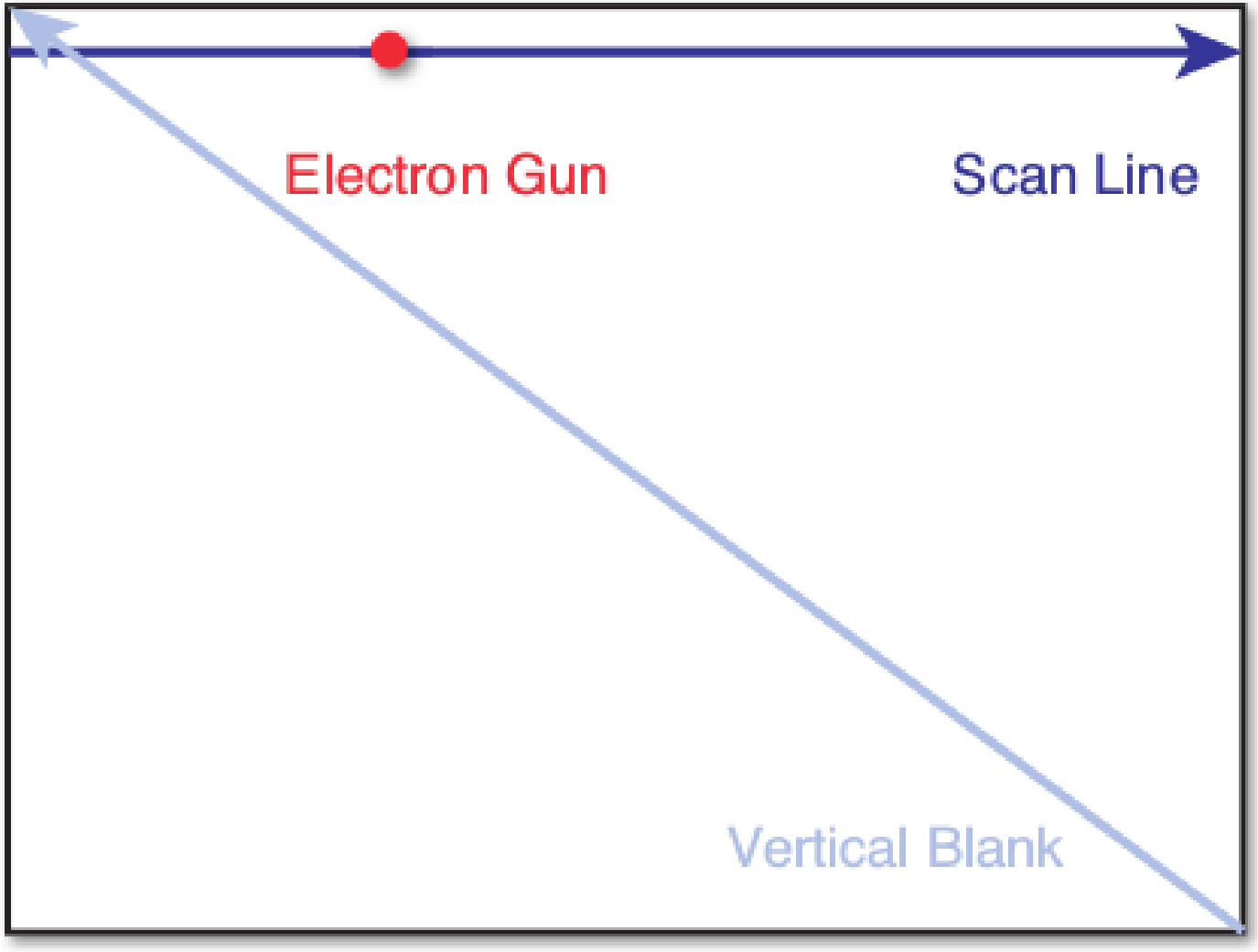
To fully understand 2D rendering, it is important to understand the limitations of display devices when these techniques were first developed. Even though we now almost exclusively use LCD or plasma displays, many of the rendering concepts that were originally developed with older monitors in mind are still in use today.

Ancient CRT monitor basics

For many years, *cathode ray tube* (CRT) displays were the predominant display technology. A CRT features an array of picture elements known as pixels. For a color display, each pixel contains a red, green, and blue sub-pixel, which can then be combined to create specific colors. The resolution of the display determines the total number of pixels. For instance, a 300x200 display would have 200 total rows, or scan lines, and each scan line would have 300 pixels, for a grand total of 60,000 pixels. The (0,0) pixel usually refers to the top-left corner, though not all displays follow this format.

In a CRT, all of the drawing is done by **an electron gun** that fires a narrow stream of electrons. This gun starts drawing in the top-left corner of the screen and shifts its aim across horizontally to draw the first scan line. It then repositions its aim so it can start all the way at the start of the subsequent scan line, repeating this process until all scan lines have been drawn.

Ancient CRT monitor basics



The vertical blank interval (VBLANK)

When the electron gun has finished drawing one frame, should be positioned at the bottom-right corner of the CRT. The amount of time it takes for the electron gun to shift its aim from the bottom-right corner all the way back to the top-left corner is known as the **vertical blank interval (VBLANK)**.

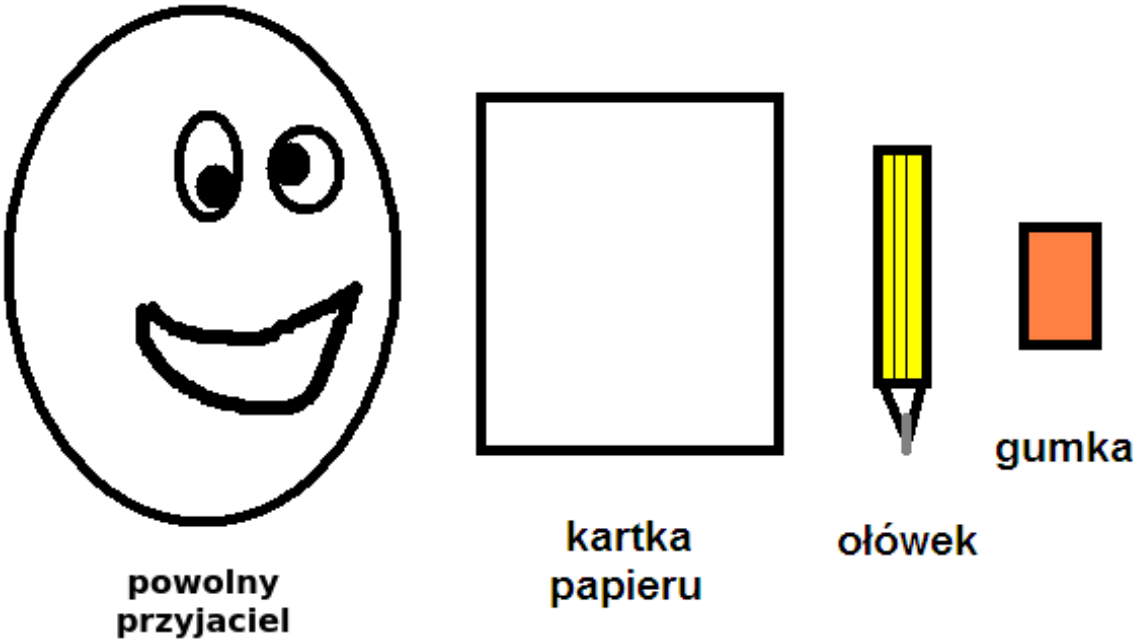
Screen tearing

First problem: no sync

Contemporary hardware featured more than enough memory to have a color buffer that could store the pixel data for the entire screen at once. This did not mean that the game loop could ignore the CRT gun entirely. Suppose the electron gun is half way through drawing the screen. At this exact time, it just so happens that the game loop hits the "generate outputs" phase. So **it starts writing the pixel information into the color buffer for the next frame, while the CRT is still drawing the previous (last) frame**. The result of this is **screen tearing**, which is when the screen shows part of two different frames at once.

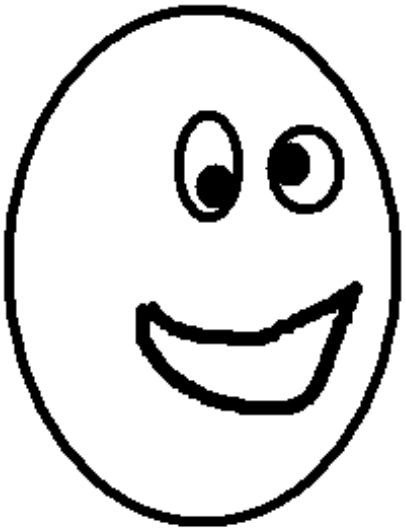
Screen tearing

Second problem: too much time consumed by frame generation



Screen tearing

Second problem: too much time consumed by frame generation



powolny
przyjaciel

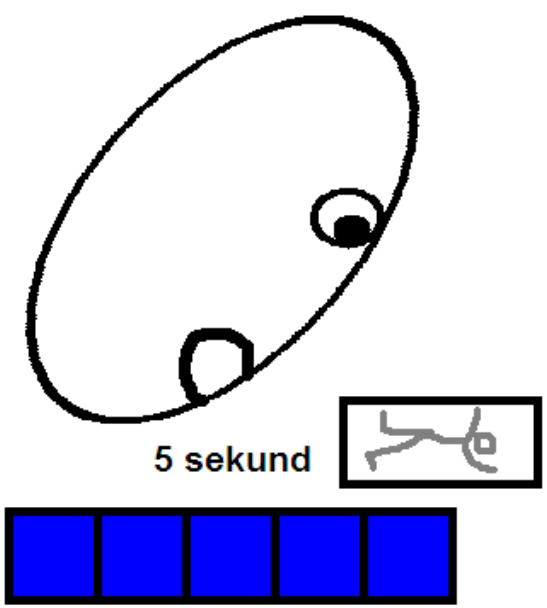
cechuje się:

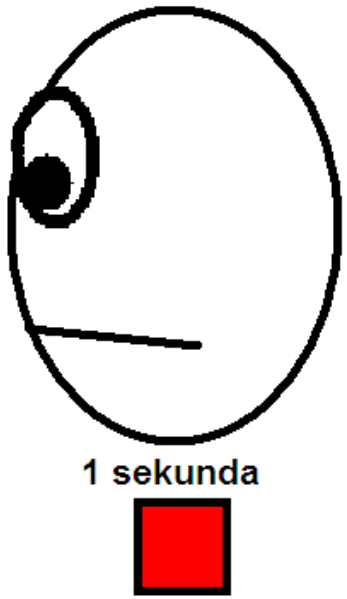
- przede wszystkim jest powolny

- zawsze ogląda obrazek 5 sekund
- po czym odwraca się na 1 sekunde
(np. spogląda za siebie by
sprawdzić czy ktoś za nim nie stoi)
- ... i kontynuuje oglądanie obrazka

Screen tearing

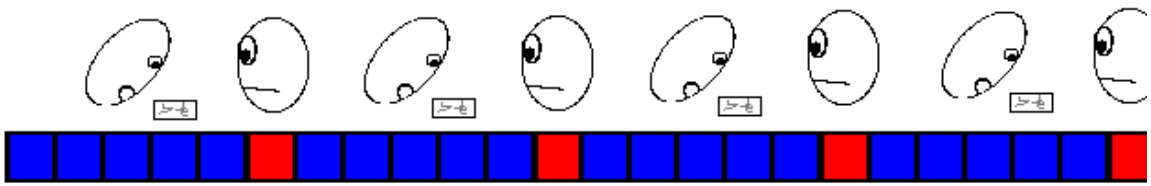
Second problem: too much time consumed by frame generation





Screen tearing

Second problem: too much time consumed by frame generation



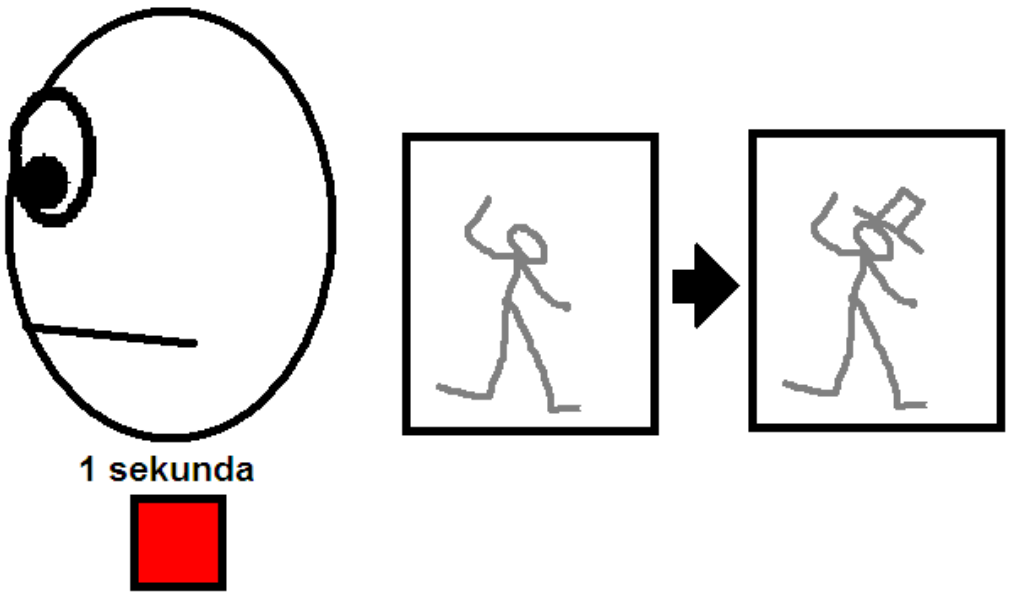
Screen tearing

Second problem: too much time consumed by frame generation

Magiczna
animowana
kartka

Screen tearing

Second problem: too much time consumed by frame generation

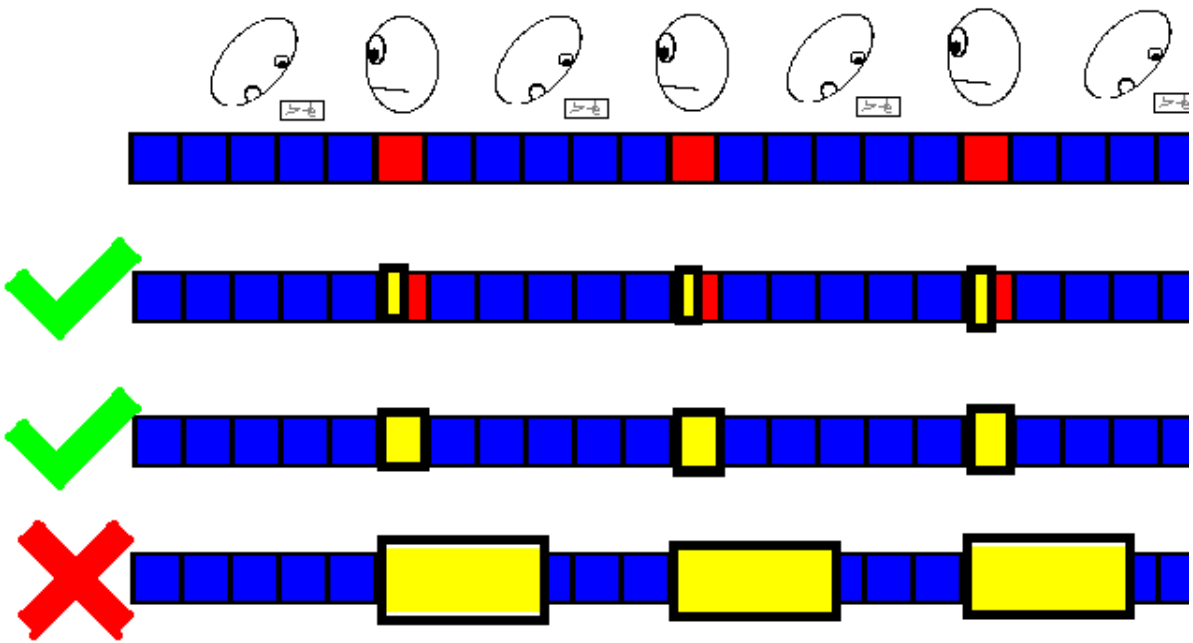


Screen tearing

Second problem: too much time consumed by frame generation

Screen tearing

Second problem: too much time consumed by frame generation

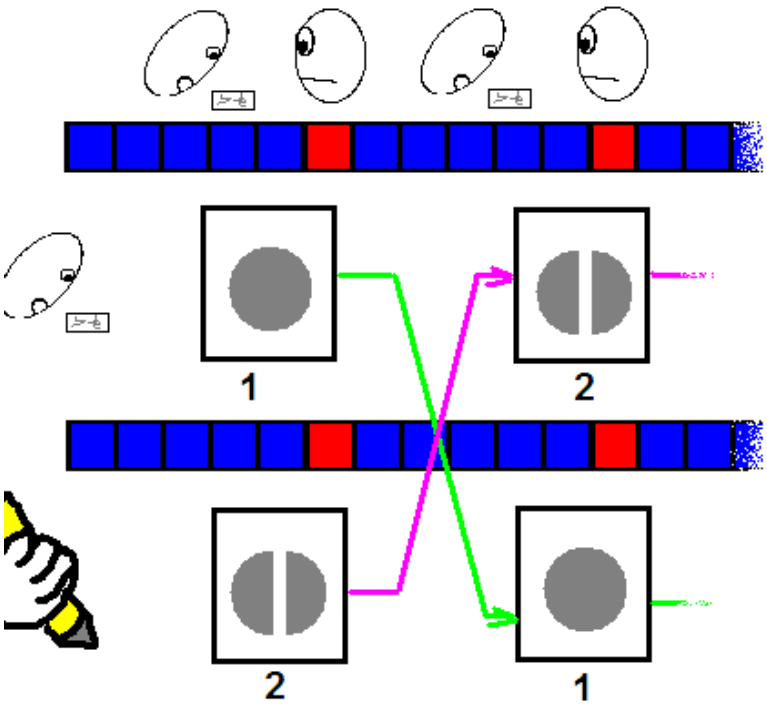


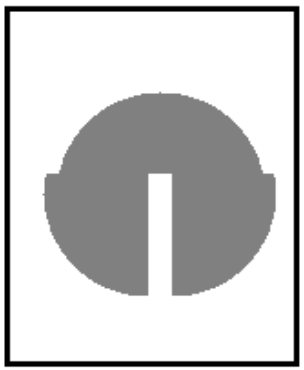
Screen tearing

Second problem: too much time consumed by frame generation

Screen tearing

Example





Screen tearing

Example

Double buffering

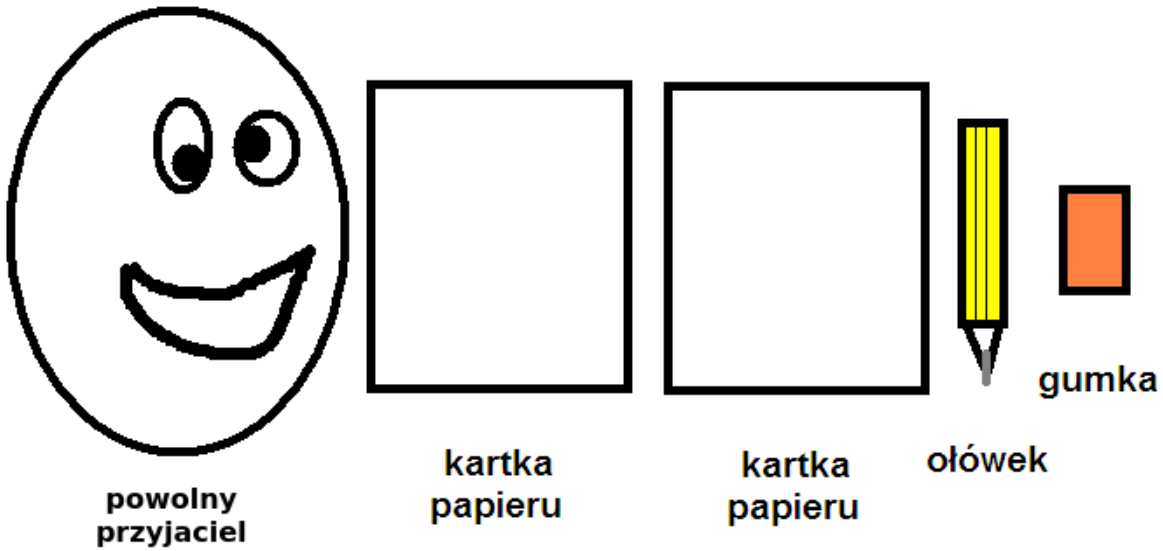
Solution for screen tearing

It is instead possible to solve screen tearing with a rendering technique called **double buffering**. In double buffering, there are two color buffers. The game alternates between drawing to these two buffers. On one frame, the game loop might write to buffer A while the CRT displays buffer B. Then on the next frame, the CRT will display buffer A while the game loop writes to buffer B. As long as both the CRT and game loop aren't accessing the same buffer at the same time, there is no risk of the CRT drawing an incomplete frame.

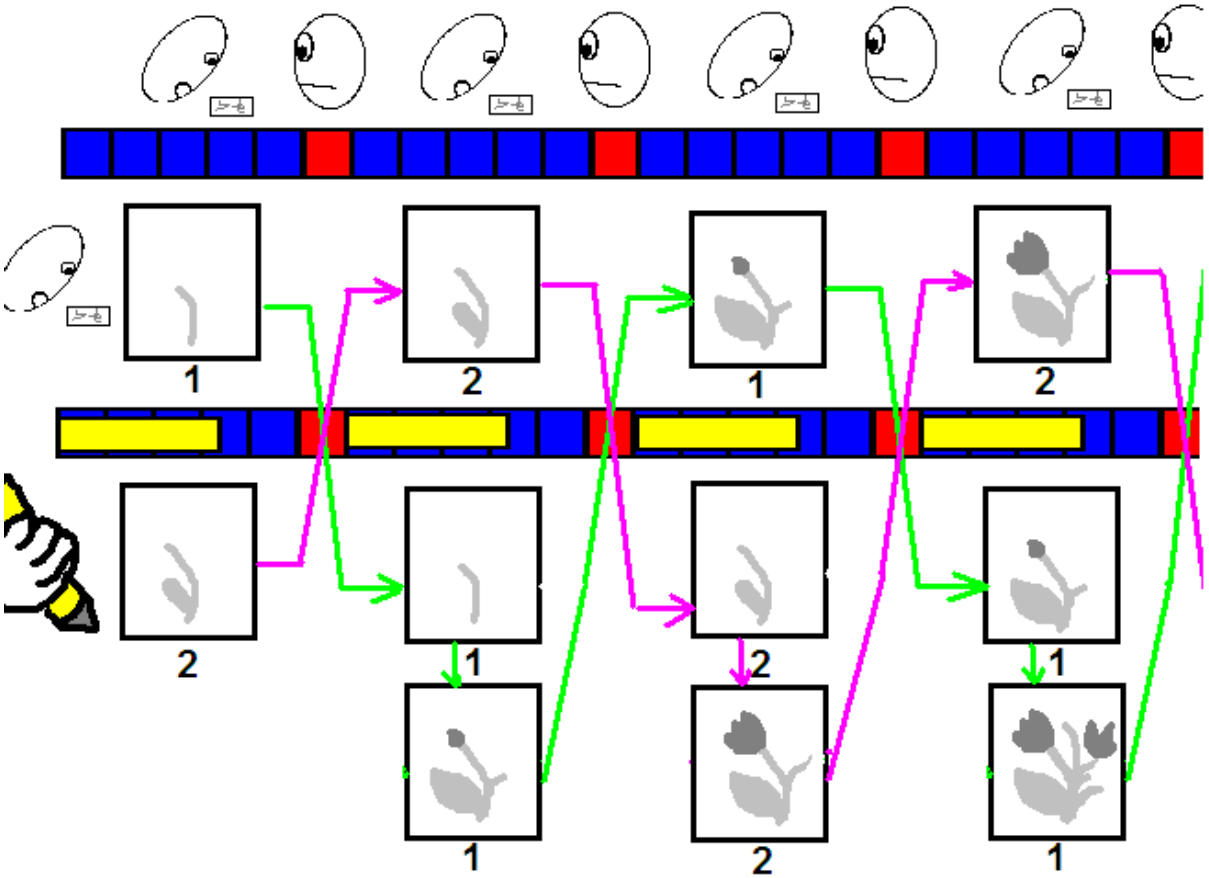
Double buffering

In order to fully prevent screen tearing, the buffer **swap must happen during VBLANK**. This is often listed as VSYNC in the graphics settings for games. In any event, because the buffer swap is a relatively fast operation, the game has a much longer period of time to render the entire frame (though ideally this should be less than the amount of time it takes the CRT to draw a frame). So long as the buffer swap occurs during VBLANK, screen tearing will be entirely avoided.

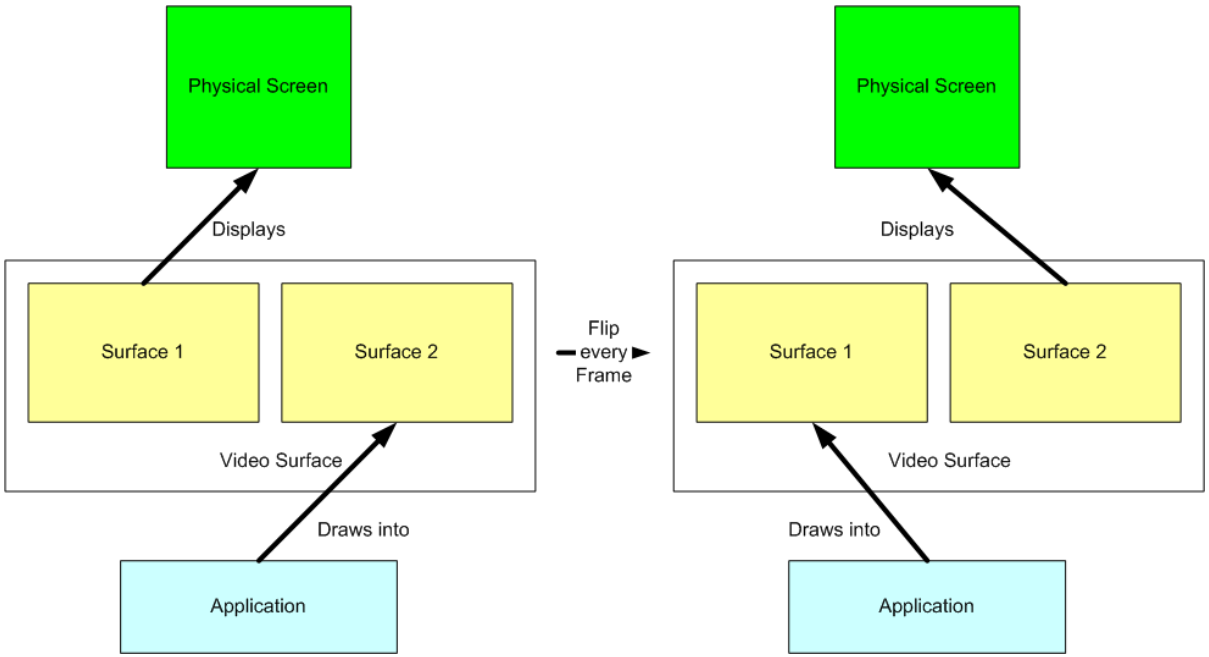
Double buffering



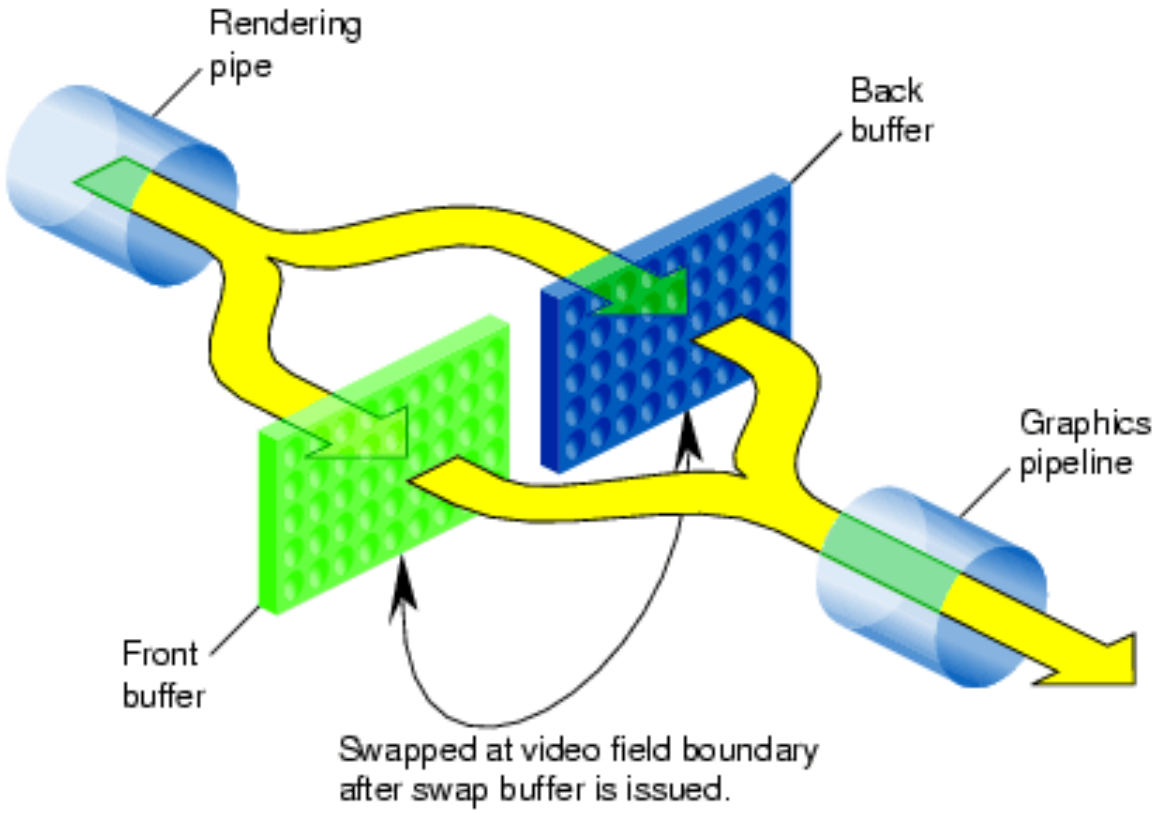
Double buffering



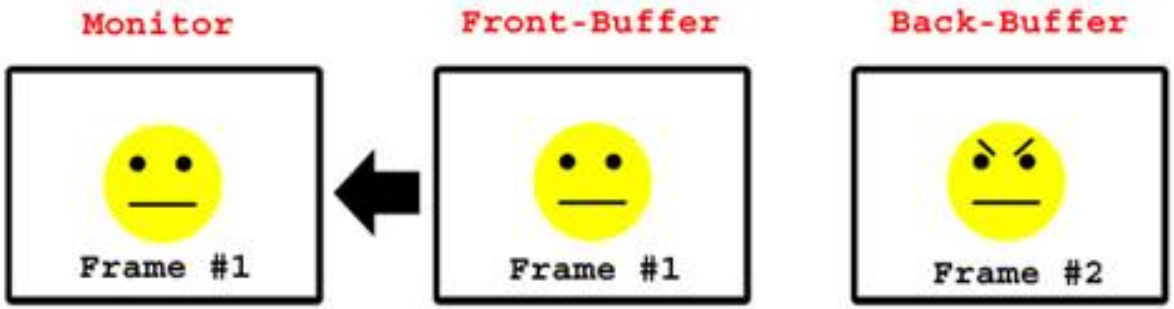
Double buffering



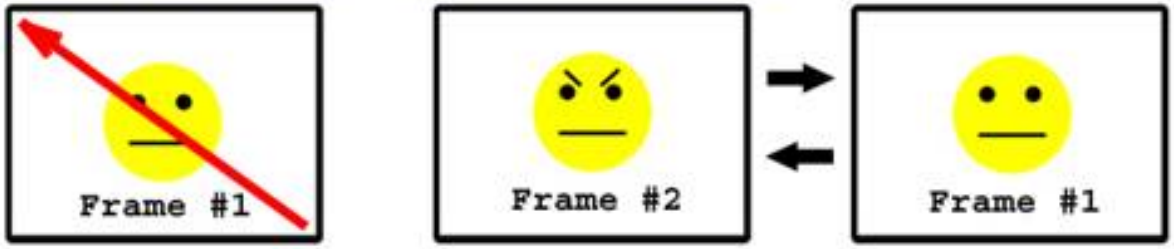
Double buffering



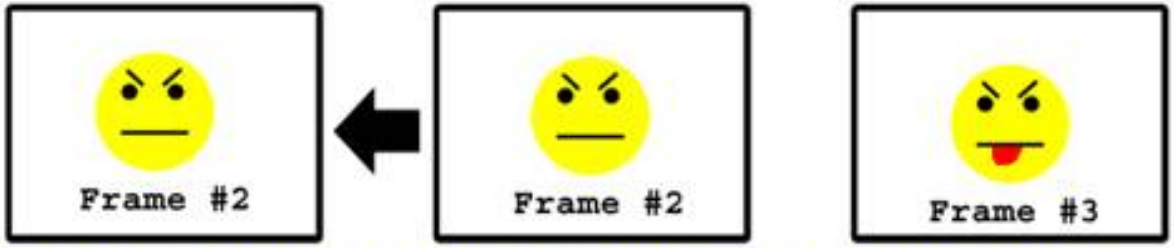
Double buffering



While the front-buffer updates the monitor,
clear and render to the back-buffer.



During the vertical retrace, swap the buffers.



Again... while the front-buffer updates the monitor,
clear and render to the back-buffer.

Screen tearing for impatient players

Some games do allow **buffer swaps to occur as soon as rendering finishes**, which means there may be some screen tearing. This is typically allowed when a user wants to run the game at a frame rate much higher than the screen refresh rate. If a particular monitor has a 60 Hz refresh rate, synchronizing the buffer swaps to VBLANK would keep the frame rate at 60 FPS. But players who are very conscientious of reducing their input lag may be able to achieve much higher frame rates if that limitation is removed.

Sprites

In computer graphics, **a sprite is a two-dimensional image or animation that is integrated into a larger scene**. Typically sprites are used to represent characters and other dynamic objects. For simple games, sprites might also be used for backgrounds, though there are more efficient approaches, especially for static backgrounds.
The term was derived from the fact that sprites, rather than being part of the bitmap data in the framebuffer, instead "floated" around on top without affecting the data in the framebuffer below, much like a ghost or "sprite". By this time, sprites had advanced to the point where complete two-dimensional shapes could be moved around the screen horizontally and vertically with minimal software overhead.

Sprites

Originally, sprites were a method of integrating unrelated bitmaps so that they appeared to be part of the normal bitmap on a screen, such as creating an animated character that can be moved on a screen without altering the data defining the overall screen. Such sprites can be created by either electronic circuitry or software. In circuitry, a hardware sprite is a hardware construct that employs custom DMA channels to integrate visual elements with the main screen in that it super-imposes two discrete video sources. Software can simulate this through specialized rendering methods.

The CPU would instruct the external chips to fetch source images and integrate them into the main screen using direct memory access channels. Calling up external hardware, instead of using the processor alone, greatly improved graphics performance. Because the processor was not occupied by the simple task of transferring data from one place to another, software could run faster; and because the hardware provided certain innate abilities, programs were also smaller.

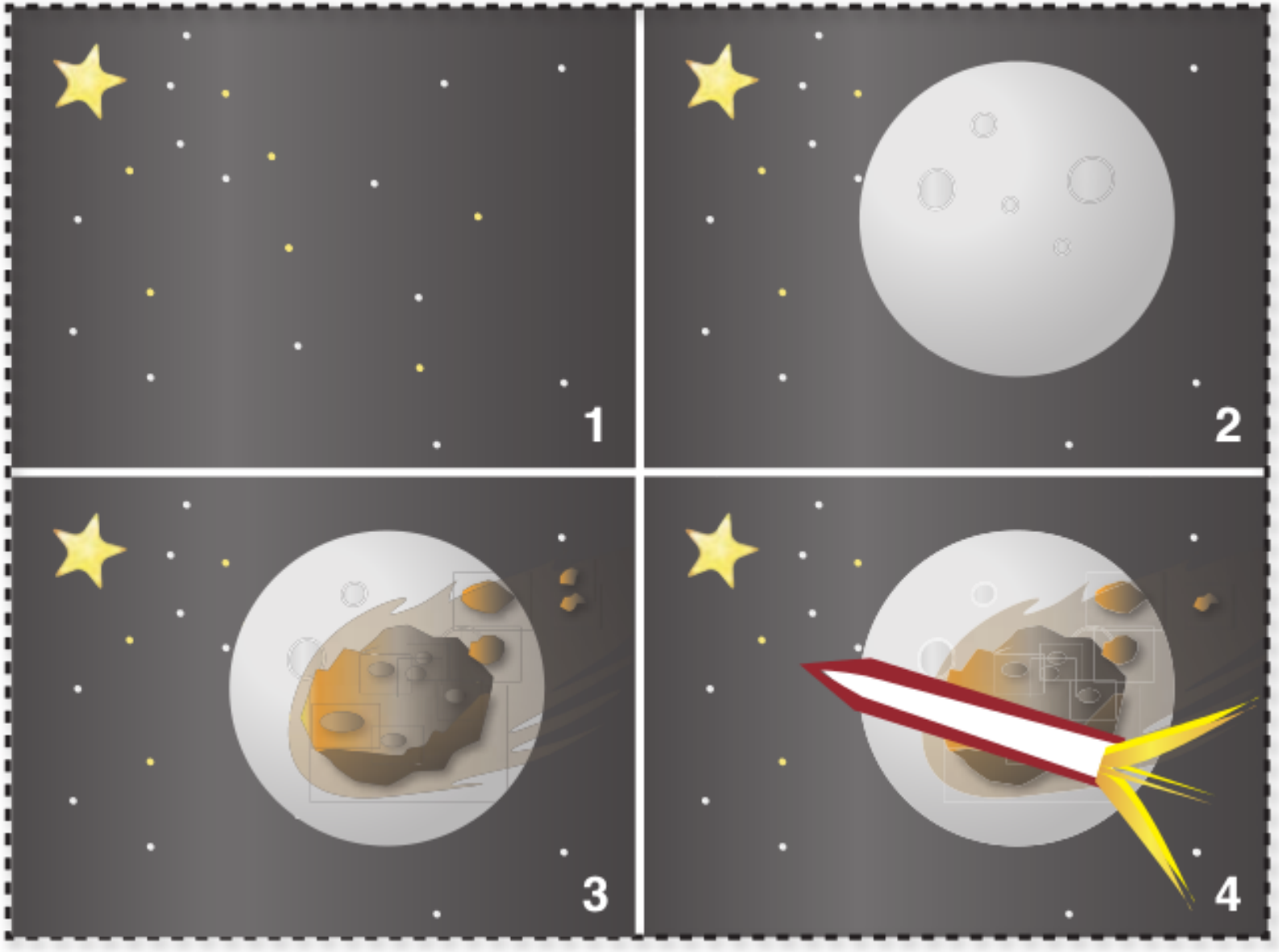
Sprites – resources

- `http://www.spriters-resource.com/`
- `http://spritedatabase.net/`
- `https://github.com/nothings/stb` – cross-platform library written in C, which can load several file formats; if not suitbale it can be a good point to start writing own code.
- `https://www.codeandweb.com/texturepacker`

Drawing sprites

Suppose you have a basic 2D scene with a background image and a character in the center. The simplest approach to drawing this scene would be to first draw the background image and then draw the character. This is much like how a painter would paint the scene on a canvas, and because of this, the approach is known as **the painter's algorithm**. In the painter's algorithm, all the sprites in a scene are sorted from back to front. When it's time to render the scene, the presorted scene can then be traversed in order so that it can be drawn appropriately.

Drawing sprites

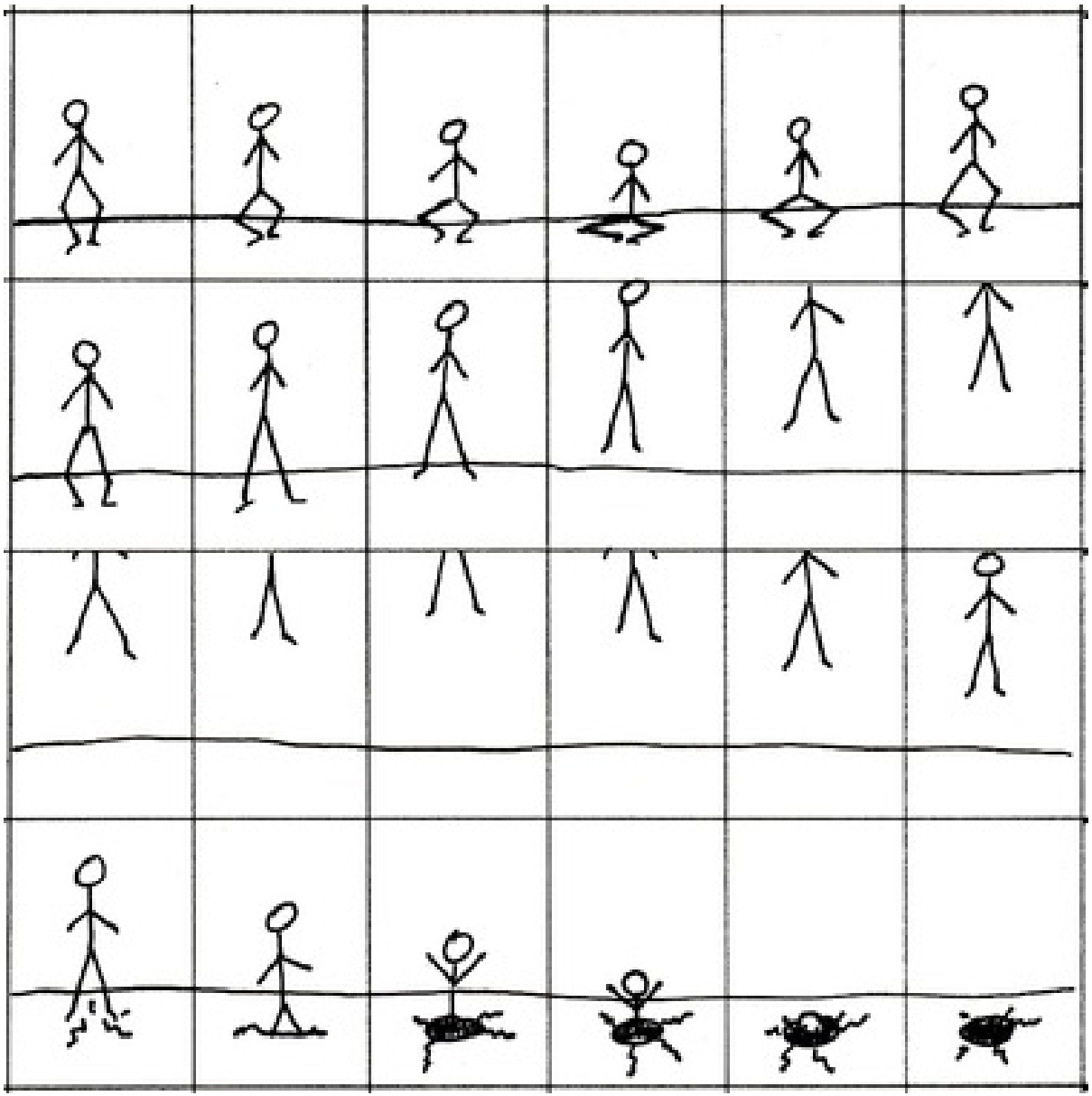


Animating sprites – flipbook

A flip book or flick book (pol. *kineograf*) is a book with a series of pictures that vary gradually from one page to the next, so that when the pages are turned rapidly, the pictures appear to animate by simulating motion or some other change.

Animating sprites – flipbook

Animating sprites – flipbook



Animating sprites

For most 2D games, animation is based on the principles of traditional flipbook animation: a series of static 2D images are played in rapid succession to create an illusion of motion.

Animating sprites

A typical approach is to have an array of images that represents all the possible states of a particular character, regardless of the particular animation. For example, a character that has both a walk and run cycle, each ten frames in length, would have an array of 20 images in total. To keep things simple, these images would be stored sequentially, which would mean frames 0–9 would correspond to the walk cycle and frames 10–19 would correspond to the run cycle.

Animating sprites – basic data structure for animation

```
struct AnimFrameData
  // The index of the first frame of an animation
  int startFrame
  // The total number of frames for said animation
  int numFrames
end
```

Animating sprites – data structure for all animations

```
struct AnimData
  // Array of images for all the animations
  ImageFile images []
  // The frame data for all the different animations
  AnimFrameData frameInfo []
end
```

Animating sprites – basic sprite code

```
class Sprite
  ImageFile image
  int drawOrder
  int x , y

  // Draw the image at the correct (x,y)
  function Draw()
end
```

Animating sprites

```
class AnimatedSprite inherits Sprite
  // All of the animation data (includes ImageFiles and FrameDat
  AnimData animData
  // The particular animation that is active
  int animNum
  // The frame number of the active animation that's being displ
  int frameNum
  // Amount of time the current frame has been displayed
  float frameTime
  // The FPS the animation is running at (24FPS by default).
  float animFPS = 24.0f

  function Initialize(AnimData myData, int startingAnimNum)
  function UpdateAnim(float deltaTime)
  function ChangeAnim(int num)
end
```

Animating sprites

```
function AnimatedSprite.Initialize(AnimData myData,
                                   int startingAnimNum)
  animData = myData
  ChangeAnim(startingAnimNum)
end
```

Animating sprites

```
function AnimatedSprite.ChangeAnim(int num)
  animNum = num
  // The active animation is now at frame 0 and 0.0f time
  frameNum = 0
  animTime = 0.0f
  // Set active image, which is just the starting frame.
  int imageNum = animData.frameInfo[animNum].startFrame
  image = animData.images[imageNum]
end
```

Animating sprites

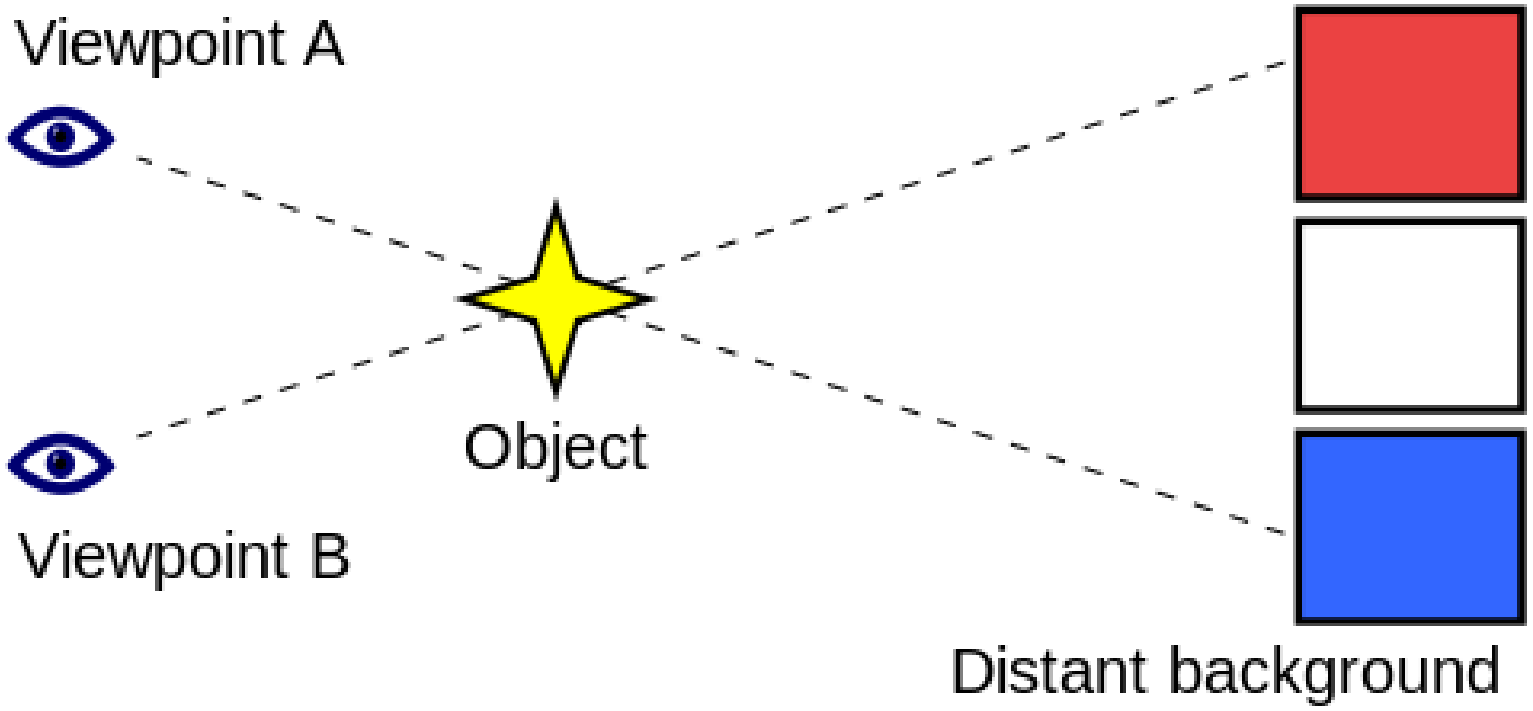
```
function AnimatedSprite.UpdateAnim(float deltaTime)
  // Update how long the current frame has been displayed
  frameTime += deltaTime
  // This check determines if it's time to change to the next frame.
  if frameTime > (1 / animFPS )
    // The number of frames to increment is the integral result of
    // frameTime / (1 / animFPS), which is frameTime * animFPS
    frameNum += frameTime * animFPS
    // Check if we've advanced past the last frame, and must wrap.
    if frameNum >= animData.frameInfo [ animNum ]. numFrames
      // The modulus (%) makes sure we wrap correctly.
      // (Eg. If numFrames == 10 and frameNum == 11, frameNum would
      // wrap to 11 % 10 = 1).
      frameNum = frameNum % animData.frameInfo[animNum].numFrames
    end
    // Update the active image.
    // (startFrame is relative to all the images, while frameNum is
    // relative to the first frame of this particular animation).
    int imageNum = animData.frameInfo [ animNum ]. startFrame + frameNum
    image = animData.images [ imageNum ]
    frameTime = fmod( frameTime , 1 / animFPS )
  end
end
```

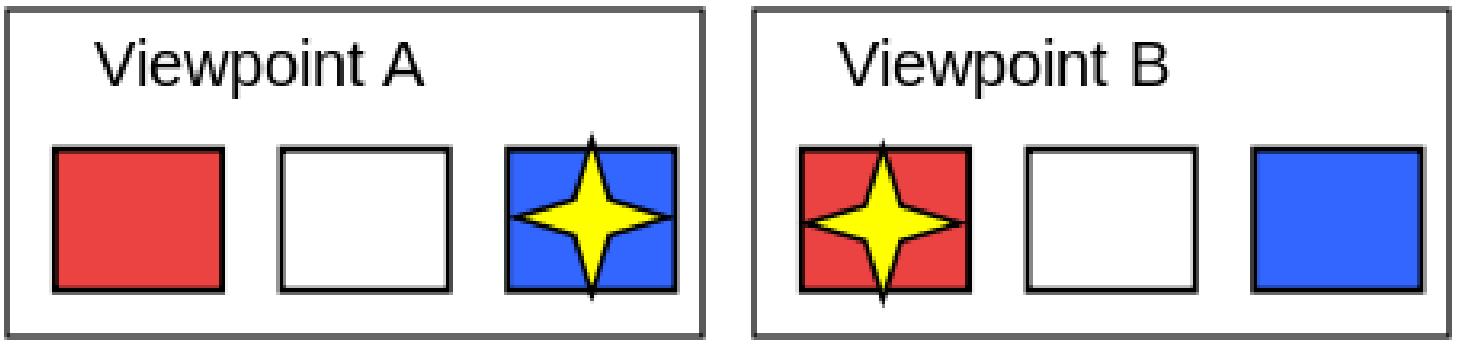
In single-axis scrolling, the game scrolls only in the x or y direction. For example, the setup code for horizontal scrolling would be as follows (see next slide).

```
const int screenWidth = 500
// All the screen-sized image backgrounds
string backgrounds [] = { "bg1.png" , "bg2.png" , /*...*/ }
// The total number of screen-sized images horizontally
int hCount = 0
foreach string s in backgrounds {
  Sprite bgSprite
  bgSprite.image .Load(s)
  // 1st screen would be x=0, 2nd x=500, 3rd x=1000, ...
  bgSprite.x = hCount * screenWidth
  bgSprite.y = 0
  bgSpriteList.Add(bgSprite)
  screenCount++
}
```

```
// camera.x is player.x as long as its clamped within the valid range
// clamping is the process of limiting a position to an area
camera.x = clamp(player.x , screenWidth / 2,
                 hCount * screenWidth - screenWidth / 2)
Iterator i = bgSpriteList.begin()
while (i != bgSpriteList.end()) {
  Sprite s = i .value()
  // find the first bg image to draw
  if ((camera.x - s.x) < screenWidth) then {
    // Image 1: s.x = 0, camera.x = 250, screenWidth/2 = 250
    // 0 - 250 + 250 = 0
    draw s at (s.x - camera.x + screenWidth /2, 0)
    // draw the bg image after this, since it might also be visible
    i++
    s = i.value()
    draw s at (s.x - camera.x + screenWidth /2, 0)
    break
  end
  i++
}
```

Parallax is a displacement or difference in the apparent position of an object viewed along two different lines of sight. The term is derived from the Greek word *parallaxis*, meaning "alteration". Nearby objects have a larger parallax than more distant objects when observed from different positions, so parallax can be used to determine distances.


Viewpoint A
Viewpoint B
Object
Distant background

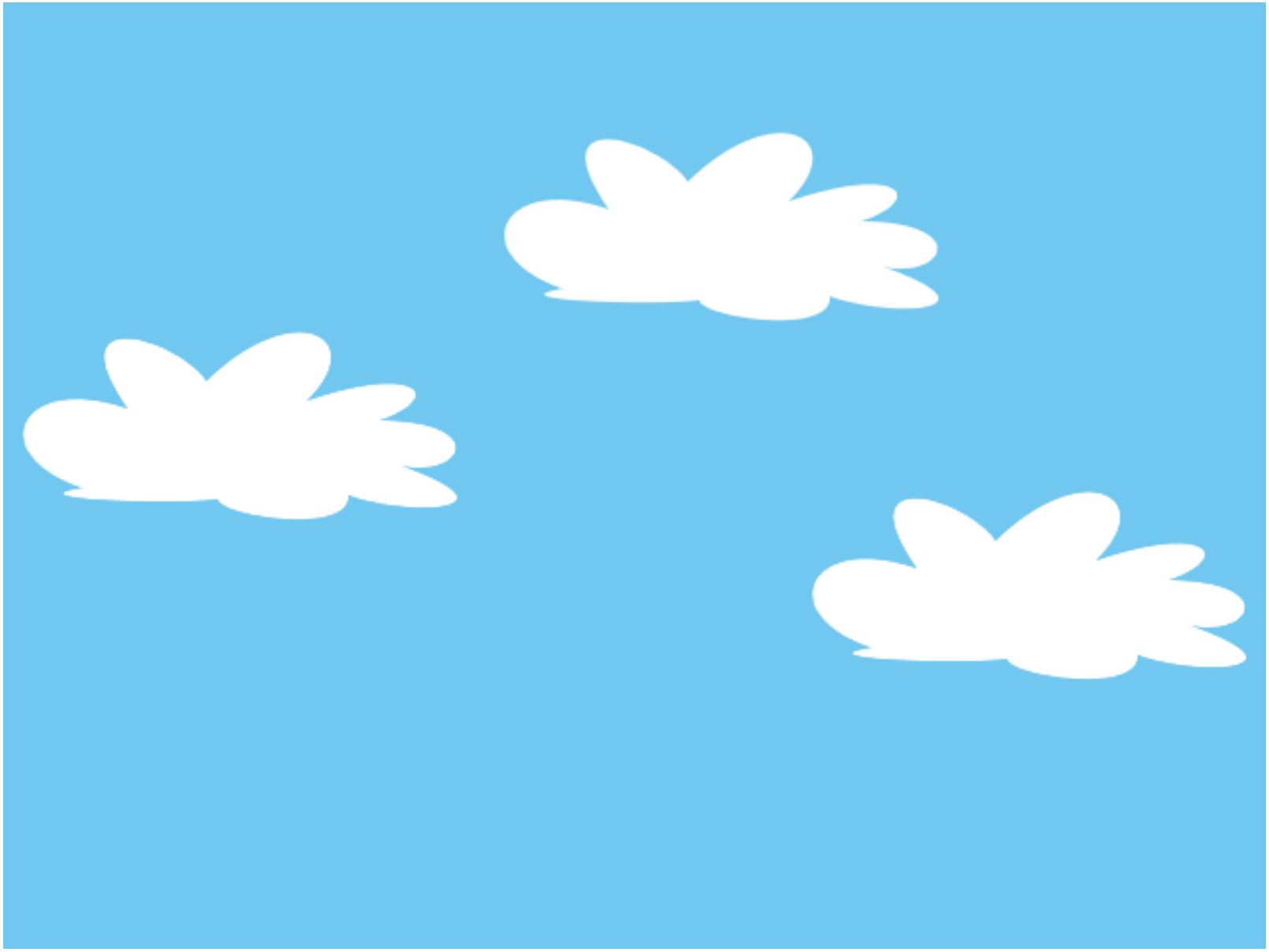

Viewpoint A
Viewpoint B

Parallax scrolling is a technique in computer graphics and web design, where background images move by the camera slower than foreground images, creating an illusion of depth in a 2D scene and adding to the immersion.

In parallax scrolling, the background is broken up into multiple layers at different depths. Each layer then scrolls at a different speed, which gives an illusion of depth. One example could be a game where there's a cloud layer and a ground layer. If the cloud layer scrolls more slowly than the ground layer, it gives the impression that the clouds are further away than the ground.

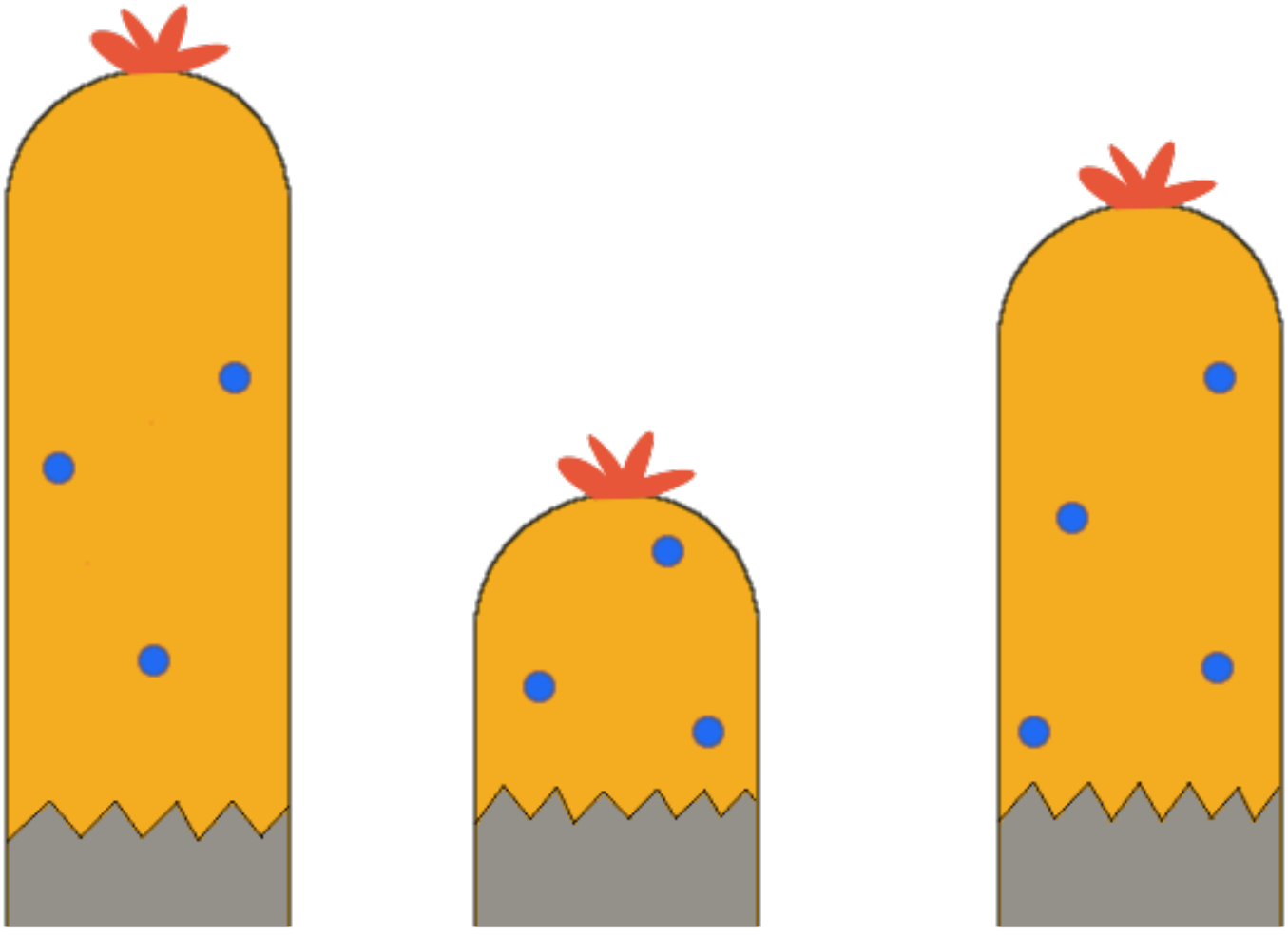
Scrolling

Parallax scrolling



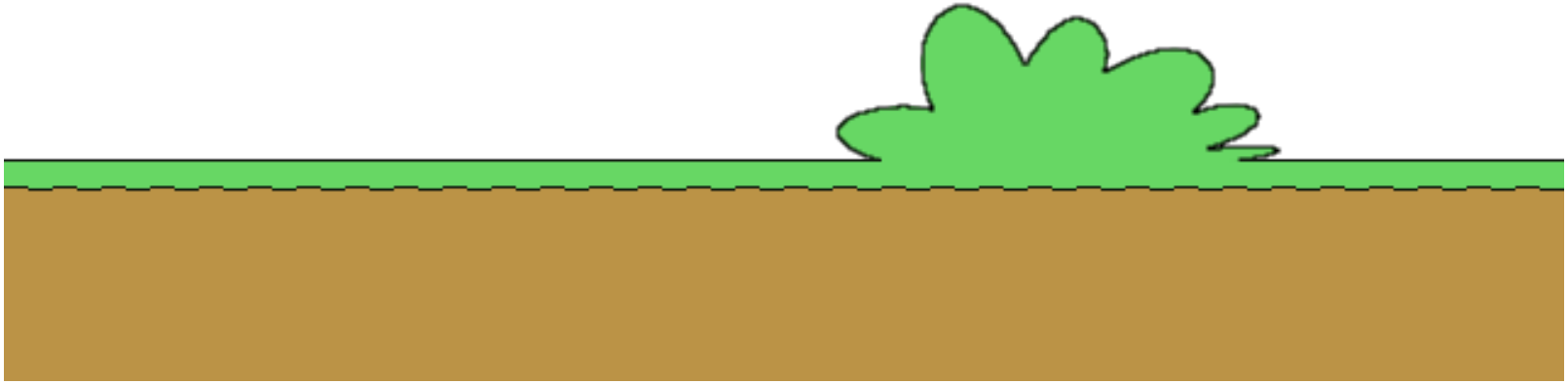
Scrolling

Parallax scrolling



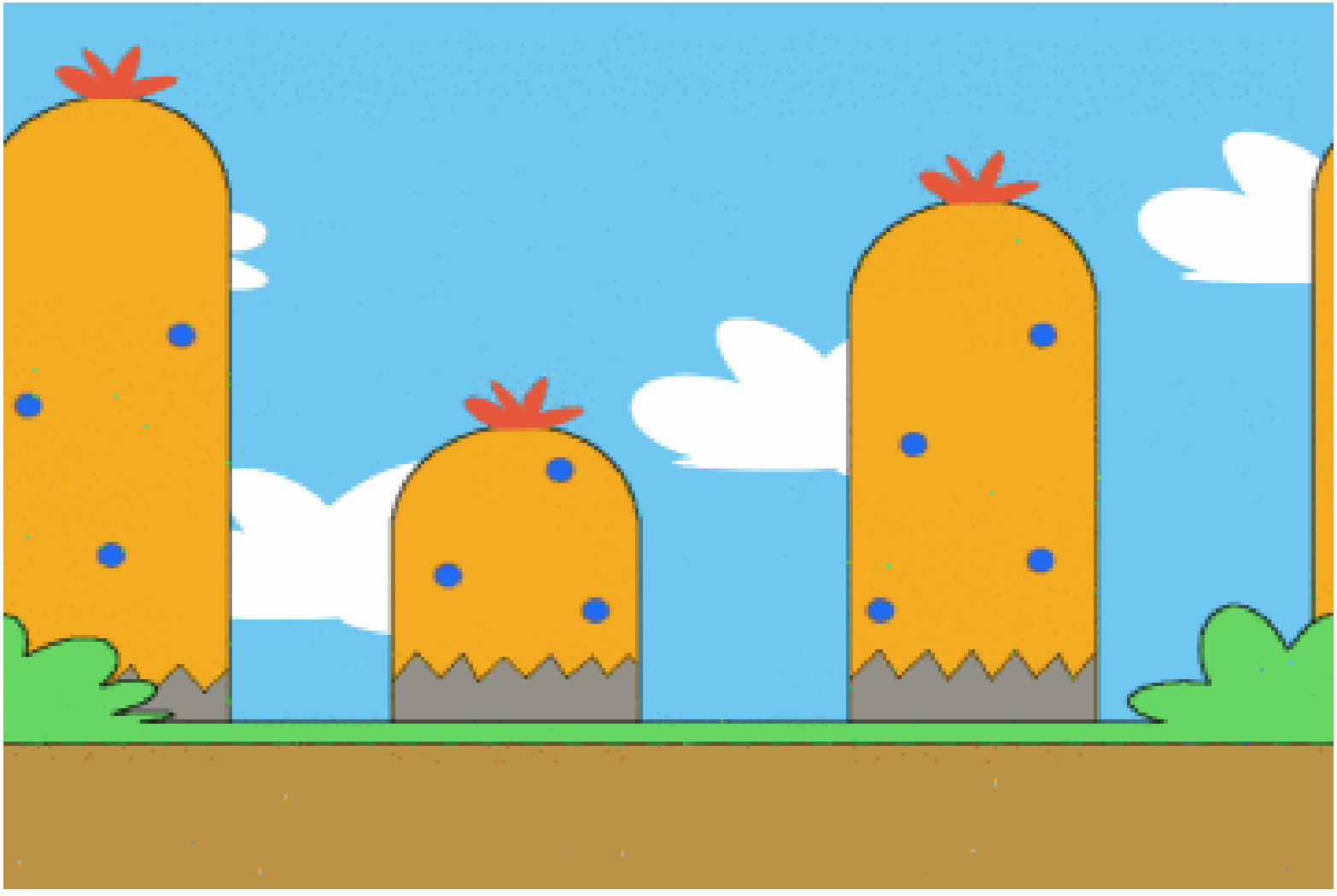
Scrolling

Parallax scrolling



Scrolling

Parallax scrolling



"Parallax scrolling example scene" by OhSqueezy - Own work. Licensed under CC BY-SA 3.0 via Commons
https://commons.wikimedia.org/wiki/File:Parallax_scrolling_example_scene.gif#/media/File:Parallax_scrolling_example_scene.gif