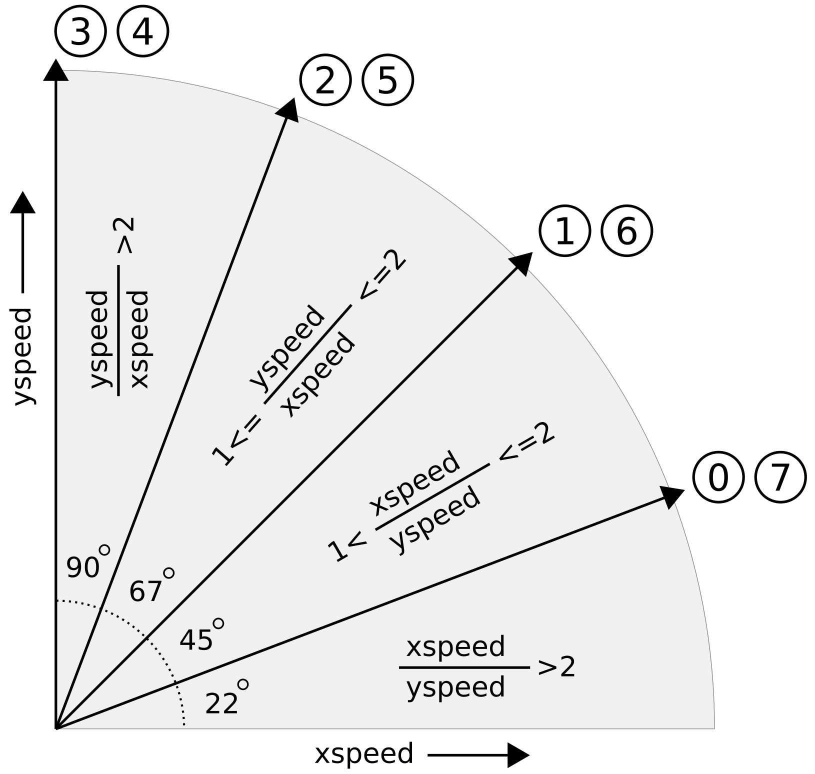
0.1 Random Tricks

Various smart techniques were employed to ensure every CPU cycle counted. This section describes random tricks used to increase game speed.

0.1.1 Bouncing Physics

When Keen throws a flower, it bounces off walls. For flat walls and floors, the bounce is easily calculated by reversing either the x-speed (for vertical walls) or the y-speed (for horizontal walls). However, handling bounces on slopes is more complex. Accurately calculating a bounce on a slope would require computationally expensive `cos` and `sin` operations.

To simplify this, the game uses an algorithm that approximates the angle to one of four values: 22°, 45°, 67° or 90°. Based on the ratio between the x-speed and y-speed, the corresponding angle and speed is calculated. The angle values 0-3 refer to a positive `xspeed`, and the values 4-7 to a negative `xspeed`.



The resulting speed is determined as a factor of either the `xspeed` or `yspeed`, depending on which of the two has the larger absolute value. For higher precision, the speed is expressed in global coordinates and therefore multiplied by 256.

```
void  PowerReact (objtype *ob)
{
    unsigned wall,absx,absy,angle,newangle;
    unsigned long speed;

    absx = abs(ob->xspeed);
    absy = ob->yspeed;

    wall = ob->hitnorth;

    [...]

    else if (wall)
    {
        ob->obclass = bonusobj;
        if (ob->yspeed < 0)
            ob->yspeed = 0;

        absx = abs(ob->xspeed);
        absy = ob->yspeed;
        if (absx>absy)
        {
            if (absx>absy*2)  // 22 degrees
            {
                angle = 0;
                speed = absx*286; // x*sqrt(5)/2
            }
            else          // 45 degrees
            {
                angle = 1;
                speed = absx*362; // x*sqrt(2)
            }
        }
        [...]        // Handle 67 and 90 degrees
    }
    if (ob->xspeed > 0)
        angle = 7-angle;
}
```

For each combination of the eight types of slopes and incoming angle, the bounce angle is determined using a simple lookup table.

Wall type	**incoming angle**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	0	0	0	0	0	0	0	0
1	7	6	5	4	3	2	1	0
2	5	4	3	2	1	0	15	14
3	5	4	3	2	1	0	15	14
4	3	2	1	0	15	14	13	12
5	9	8	7	6	5	4	3	2
6	9	8	7	6	5	4	3	2
7	11	10	9	8	7	6	5	4

Figure 1: Bounce lookup table `bounceangle[8][8]`.

Each entry in the table corresponds to one of 16 possible bounce angles. For example, when a wall type 3 slope is hit with an incoming angle of 22° and postive `xspeed`, the lookup table refers to bounce angle #5. Each bounce angle is decomposed into a new `xspeed` and `yspeed`.

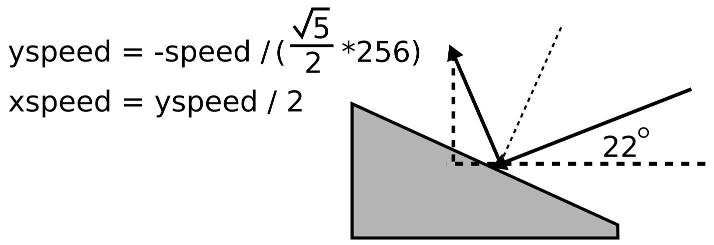

$\text{yspeed} = -\text{speed} / (\frac{\sqrt{5}}{2} * 256)$
$\text{xspeed} = \text{yspeed} / 2$
22°

Figure 2: Walltype 3 with incoming angle of 22°.

```
speed >>= 1;                  // speed / 2 after bounce
newangle = bounceangle[ob->hitnorth][angle];
switch (newangle)
{
[...]

case 5:
  ob->yspeed = -(speed / 286);
  ob->xspeed = ob->yspeed / 2;
  break;

[...]
}
```

It is worth noting that in some cases, the resulting bounce angle does not follow the laws of physics. For instance, an incoming angle of 22° on a 45° slope results in a bounce angle of 90° instead of the expected 67°.

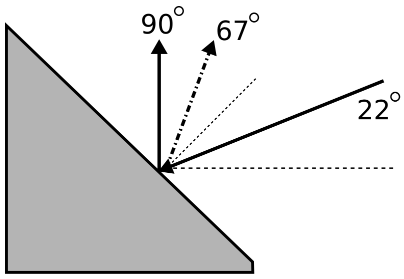


Figure 3: Incoming angle of 22° results in bounce angle of 90°.

0.1.2 Screen fades

When a new level is loaded, the screen fades from black to the default colors by reassigning the color palette. Each of the 16 color indices can be reprogrammed to any "RGBI" color, simply by calling the BIOS software interrupt `10h`.

```
_AX = 0x1000 ; Set One Palette Register
_BL = 0      ; index color number to set
_BH = 0x5    ; 6-bit rgbRGB color to display for that index
geninterrupt (0x10) ; Generate Video BIOS interrupt
```

By using `_AX=1002h`, the entire palette can be reprogrammed at once. In this process, `ES:BX` points to 17 bytes, where each byte represents an RGBI value for one of the 16 palette indices, plus one for the border.

#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1	2	3	4	5	6	7	0
2	0	0	0	0	0	0	0	0	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0
3	0	1	2	3	4	5	6	7	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0
4	0	1	2	3	4	5	6	7	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0
5	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f	0x1f

Figure 4: Color fading table `colors[7][17]`.

Fading the screen from black to color is straightforward.

```
void VW_FadeIn(void)
{
    int i;

    for (i=0;i<4;i++)
    {
        colors[i][16] = bordercolor;
        _ES=FP_SEG(&colors[i]);
        _DX=FP_OFF(&colors[i]);
        _AX=0x1002;
        geninterrupt(0x10);
        VW_WaitVBL(6);
    }
    screenfaded = false;
}
```