2D Games

Fundamentals of Game Development

Instructor : Dr. Behrouz Minaei (b_minaei@iust.ac.ir)
Teaching Assistant : Morteza Rajabi (mtz.rajabi@gmail.com)

Data Structures

- A way to encode the character graphics
- A way to encode background images
- A way to store the game map

Sprite

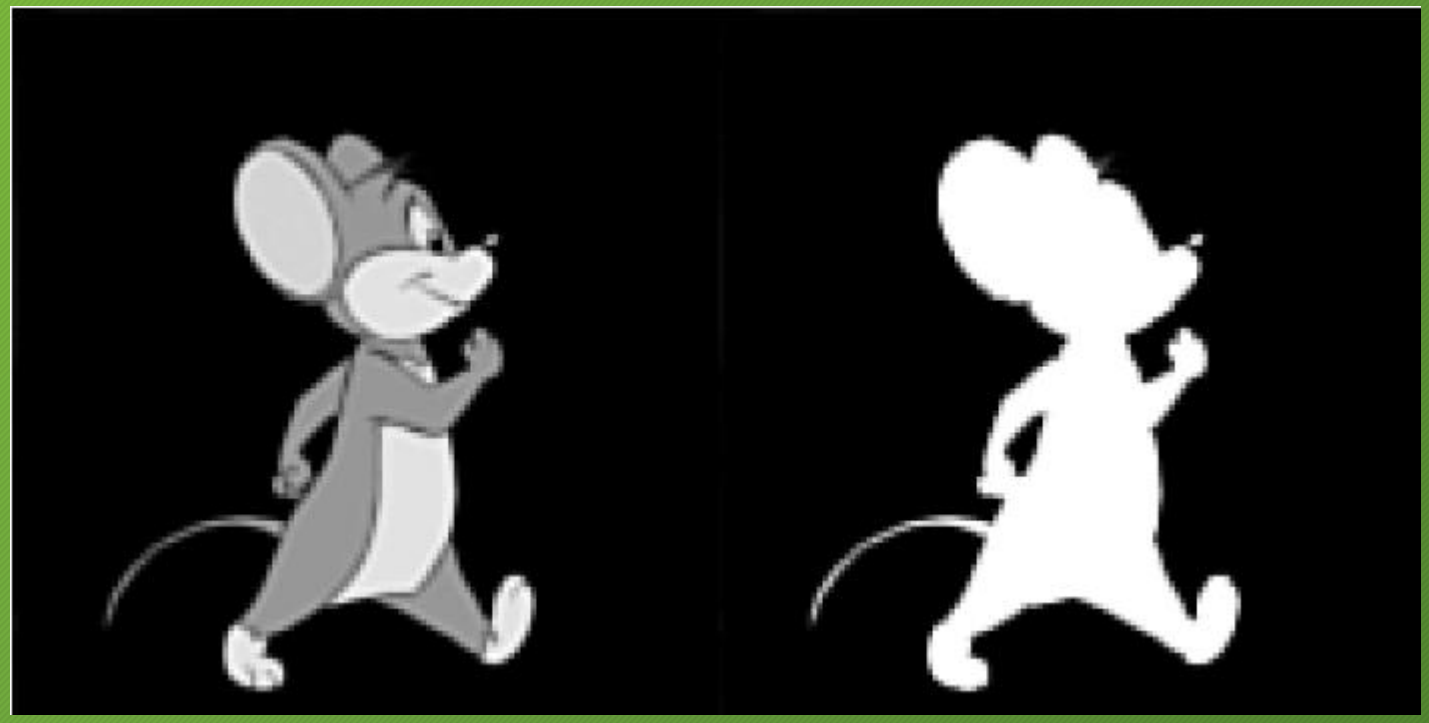
- 2D games store each character in a rectangular, bitmapped graphic, so it can be drawn quickly onscreen.
- Additional information allows the game code to detect transparent areas, so our characters blend seamlessly with the backgrounds.
- Then, specific hardware and software is used to render these layers to the screen. Each of these bitmaps (with its associated transparency information) is called a *sprite*..

Sprite Transparency

- we need to talk about transparency. Our sprites need to blend seamlessly with the background, so we need a way to encode the parts of the sprite that should leave the background unaffected.
- Several approaches have been used to achieve this effect.
- One popular alternative is to use a separate 1-bit mask to encode transparent zones.
- This mask is used in the transfer process to ensure the blend is performed properly. The mask is multiplied by the background, leaving in black the area that will later be occupied by the sprite.

Transparency

- Then, by adding the sprite to the background directly, the blending is achieved.

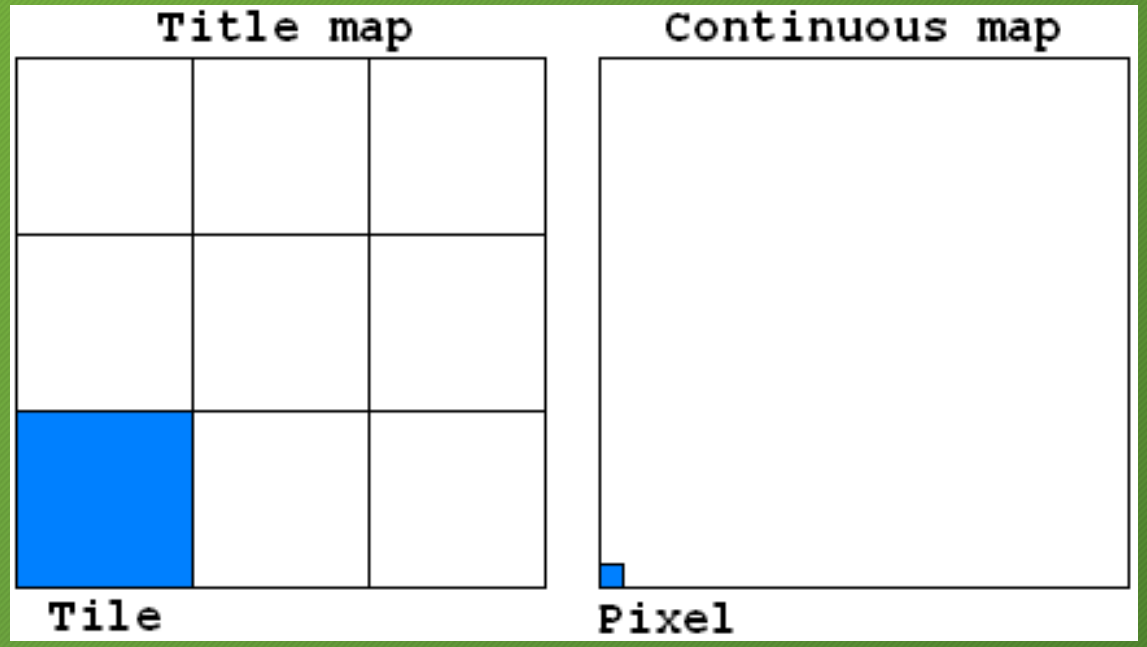


Mapping Matrices

- Mapping is a compression technique that will allow us to create good-looking game worlds at a fraction of the memory footprint.
- We will lose some visual variety in the process, but our dream game will fit on our target platform.
- Mapping is an extremely popular technique. It was used in thousands of games for classic consoles and arcades, and is still used today, even in some 3D games.
- The key idea is to divide our game world into a set of tiles or basic primitives. Each tile will represent a rectangular pattern, which we will combine with other tiles to represent the level.

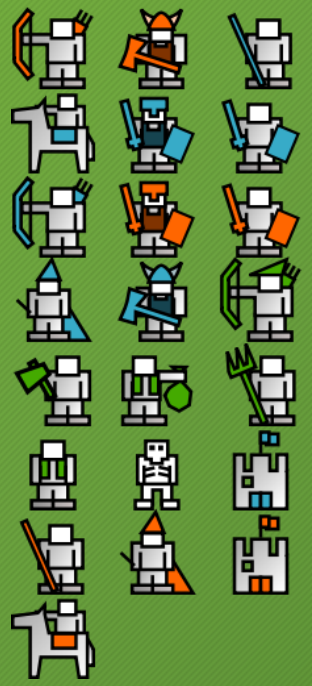
Mapping Matrices

- We have two kinds of maps: continues maps and tile maps
- The building block in continues maps is **Pixel**
- The building block in tile maps is **tiles**

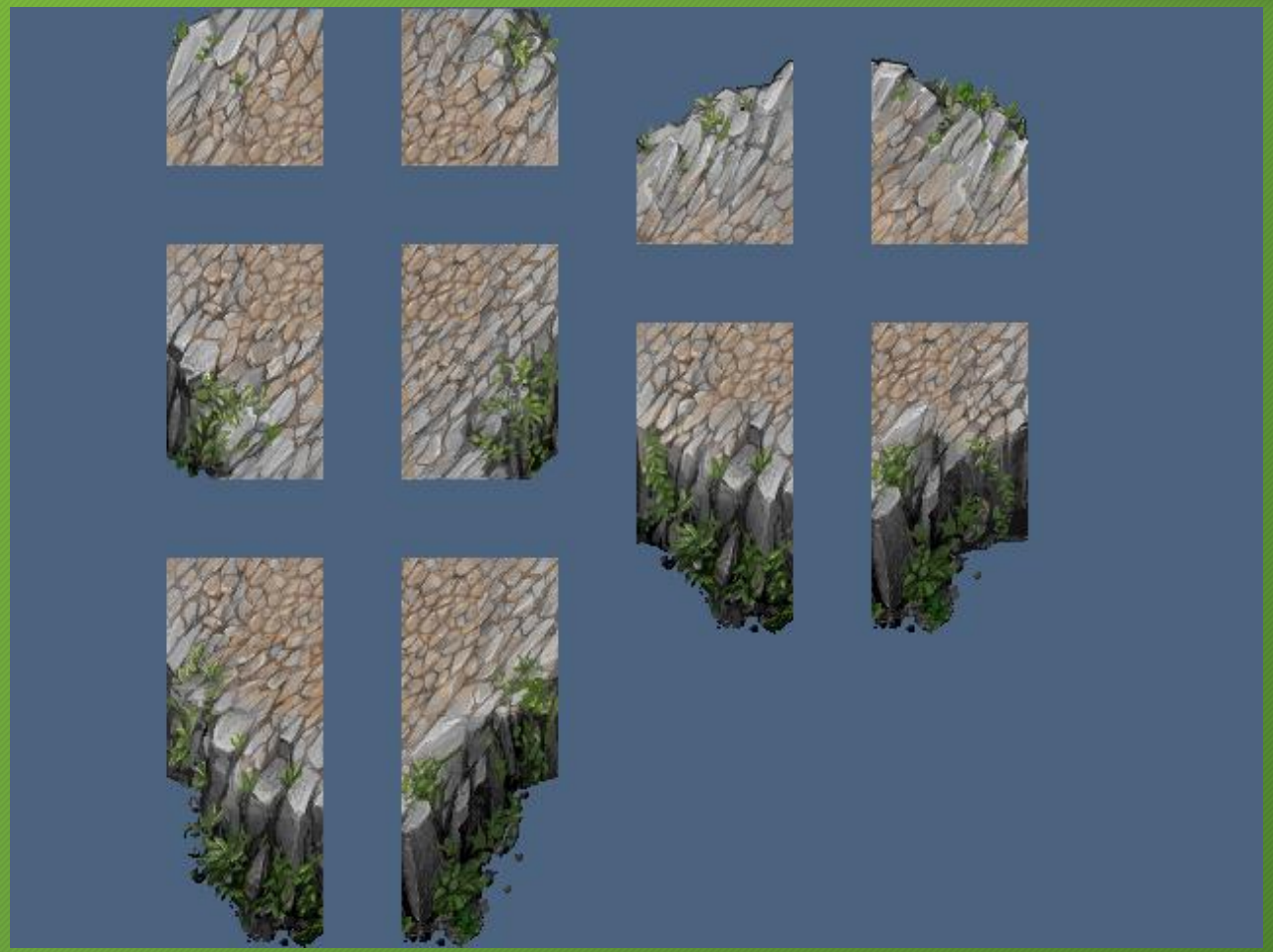


Mapping Matrices (Tile Set)

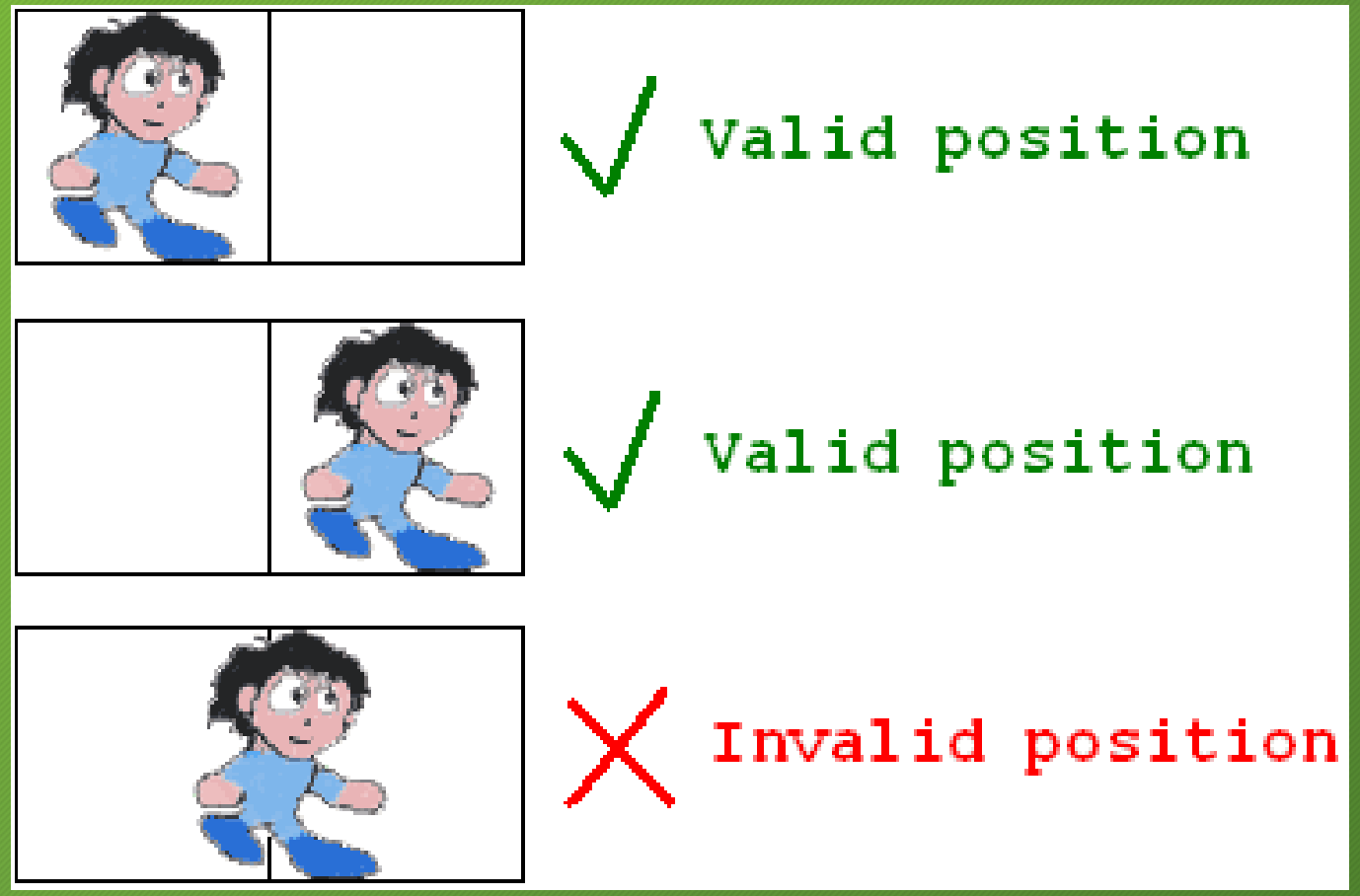
- So, if our game world must represent grass, stone, snow, and sand, we will use four tiles, and then map them as if we were putting tiles on the floor



This beautiful scene is consists of Tiles too!



Valid position vs Invalid position



Formula

- there is an equation that will give us the memory size of a single tile, as a function of several parameters. Keep this in mind when designing your game so your graphics fit in memory:

$$\text{Size} = \text{bits per pixel} * \text{wide} * \text{tall}$$

Real Time Loops

- The most popular alternative for those platforms that do not support a solid concurrency mechanism is to implement threads using regular software loops and timers in a single-threaded program.
- The key idea is to execute update and render calls sequentially, skipping update calls to keep a fixed call rate.
- We decouple the render from the update routine. Render is called as often as possible, whereas update is synchronized with time.

2D Games Algorithmes

Screen Based Games

- The simplest mapped game is the screen-based game, in which the player confronts a series of screens.
- When he exits one screen through its edge, the graphics are substituted by those in the next screen, and so forth.
- There is no continuity or transition between screens.
- In these games, each screen is represented using a different mapping matrix, which represents the screen layout of the different elements.
- So, for a *320x240 screen using 32x32 tiles, we would store the screen in a 10x8 matrix of bytes.*

Screen Based Games

- Then, the rendering routine is very straightforward. Its code would look something like this:

```
#define tile_wide 32
#define tile_high 32
#define screen_wide 320
#define screen_high 240

int xtiles=screen_wide/tile_wide;
int ytiles=screen_high/tile_high;

for (yi=0;yi<ytiles;yi++)
    {
    for (xi=0;xi<xtiles;xi++)
        {
        int screex=xi*tile_wide;
        int screey=yi*tile_high;
        int tileid=mapping_matrix [yi][xi];
        blit(tile_table[tileid],screenx,screeny);
        }
    }
```

Two-way and Four-way Scrollers

- Screen-based games divide the gameplay into chunks that must be dealt with individually. That makes rendering easier, at the cost of breaking the entertainment each time screens are changed.
- It would be much better, in terms of gameplay, if the action was a continuum, with no screen swapping at all.
- This is the goal of scrolling games: to create a larger-than-screen game world that we can continually explore from a sliding camera.

Rendering algorithm

- Scrolling games are a bit more complex than screen-based games. The game world is larger than the player's screen, so we need to process only those areas that are visible to the player.
- Besides, the screen mapping of a scrolling world depends on the player's position, making our code significantly harder.
- The initial problem we will be dealing with will be selecting the "window" or portion of the game world that will be visible.

Rendering algorithm

```
#define tile_wide 32
#define tile_high 32
#define screen_wide 320
#define screen_high 240
```

- The Number of on screen tiles are:

```
int xtiles=(screen_wide/tile_wide)+1;
int ytiles=(screen_high/tile_high)+1;
```

- Assuming the player is at

```
Int playerx;
int playery;
```

- His Cell position will be at

```
tileplayerx= Playerx/tile_wide
tileplayery= Playery/tile_high
```

The Game Logic Section- World Update

- And we must paint the following interval

```
X: ( tileplayerx – xtiles/2 .... tileplayerx + xtiles/2)
Y: ( tileplayery – ytiles/2 .... tileplayery + ytiles/2)
```

- Now we know which tiles in the mapping matrix are visible and should be painted.
- We must now calculate where in screen space those tiles should be painted. In a screen-based game, we would project at

```
int screenx=xi*tile_wide;
int screeny=yi*tile_high;
```

The Game Logic Section- World Update

- which means we are painting the screen as if it was a checkerboard.
- Now, the checkerboard is larger than the screen, and it slides according to the player's position. Thus, the new transform will be

```
int screenx=xi*tile_wide - playerx;
int screeny=yi*tile_high - playery;
```

- If we implement the previous transform "as is," we are translating by the player's position, which would mean the player is located in the coordinates 0,0 in screen space.
- This is not usually the case, as players in scrollers are usually placed in the center of the screen. Clearly, the player's coordinates are

```
Screenplayerx=screenx/2
Screenplayery=screeny/2
```

The Game Logic Section- World Update

- So the final correct world to screen transformation is as follows

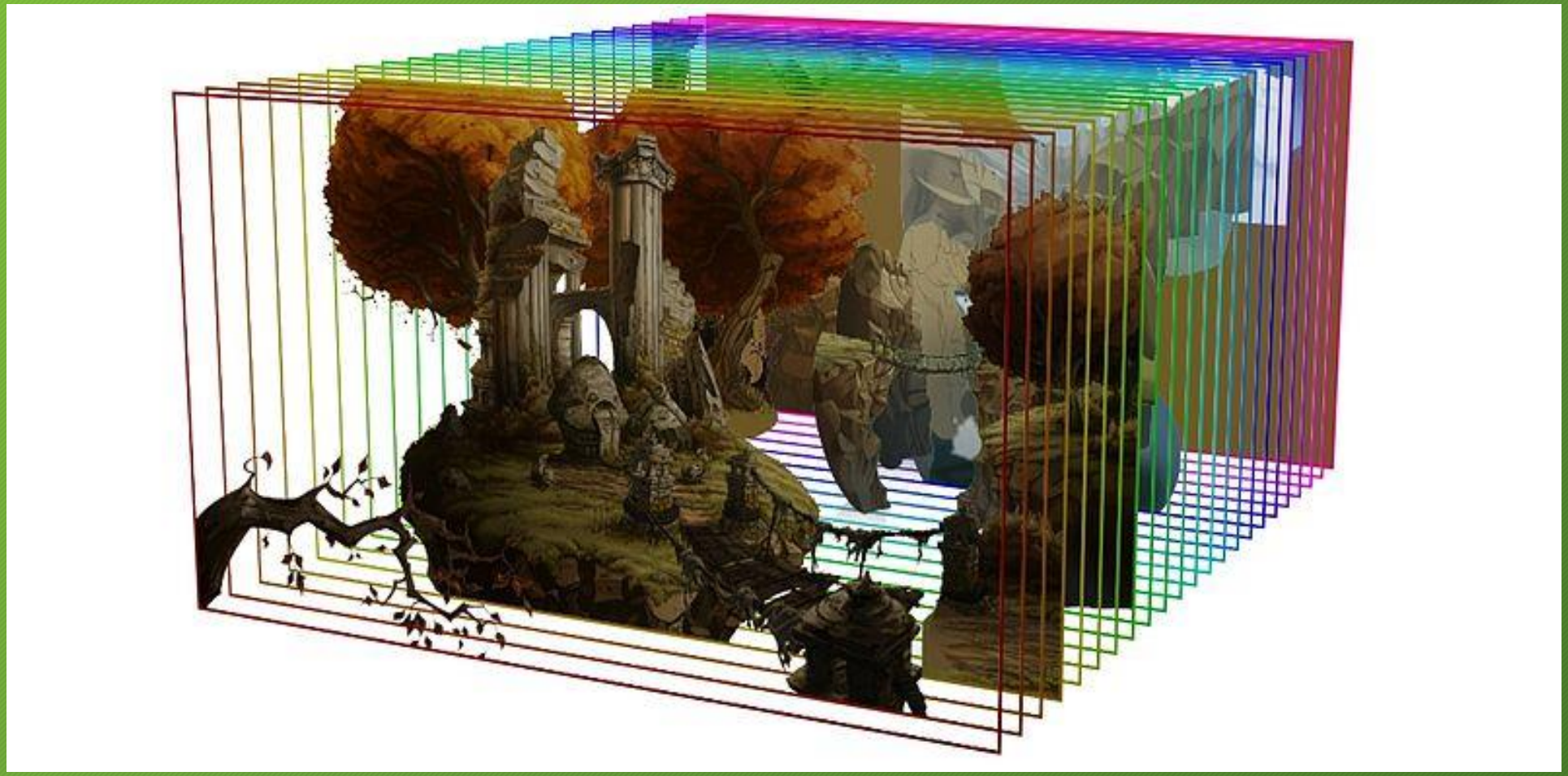
```
int screenx=xi*tile_wide – playerx-screenplayerx;
```

```
int screeny=yi*tile_high – playery-screenplayery;
```

- And the final render routine is

```
#define tile_wide 32
#define tile_high 32
#define screen_wide 320
#define screen_high 240

int beginx= tileplayerx – xtiles/2;
int beginy= tileplayery – ytiles/2;
int endx= tileplayerx + xtiles/2;
int endy= tileplayery + ytiles/2;

tileplayerx= Playerx/tile_wide
tileplayery= Playery/tile_high

int xtiles=screen_wide/tile_wide;
int ytiles=screen_high/tile_high;

for (yi=beginy;yi<endy;yi++)
    {
    for (xi=beginx;xi<endx;xi++)
            {
            int screenx=xi*tile_wide – playerx-screenplayerx;
            int screeny=yi*tile_high – playery-screenplayery;
            int tileid=mapping_matrix [yi][xi];
            blit(tile_table[tileid],screenx,screeny);
            }
    }
```

Multi Layered 2D Games



Rendering: Multi Layer

- Multilayered engines use several mapping matrices to encode the game map. One matrix represents the background, another matrix the trees, and so on. Then, matrices are painted in a parallel fashion, such as:

```
for (yi=beginy;yi<endy;yi++)
    {
    for (xi=beginx;xi<endx;xi++)
        {
        int screenx=xi*tile_wide – playerx-screenplayerx;
        int screeny=yi*tile_high – playery-screenplayery;
        for (layeri=0;layeri<numlayers;layeri++)
            {
            int tileid=mapping_matrix [layeri][yi][xi];
            blit(tile_table[tileid],screenx,screeny);
        }
    }
```

Parallax Scrollers

- Parallax is the optical phenomenon described as the apparent displacement of a distant object (with respect to a more distant background) when viewed from two different positions.
- In other words, if you are driving a car on a highway and you see a fence along the side with mountains on the horizon, the fence seems to move much faster than the mountains.
- This is due to the fact that perceived object size decreases with the distance to the viewer squared. Thus, apparent movement speeds from distant (but large) objects are smaller than speeds from foreground objects.

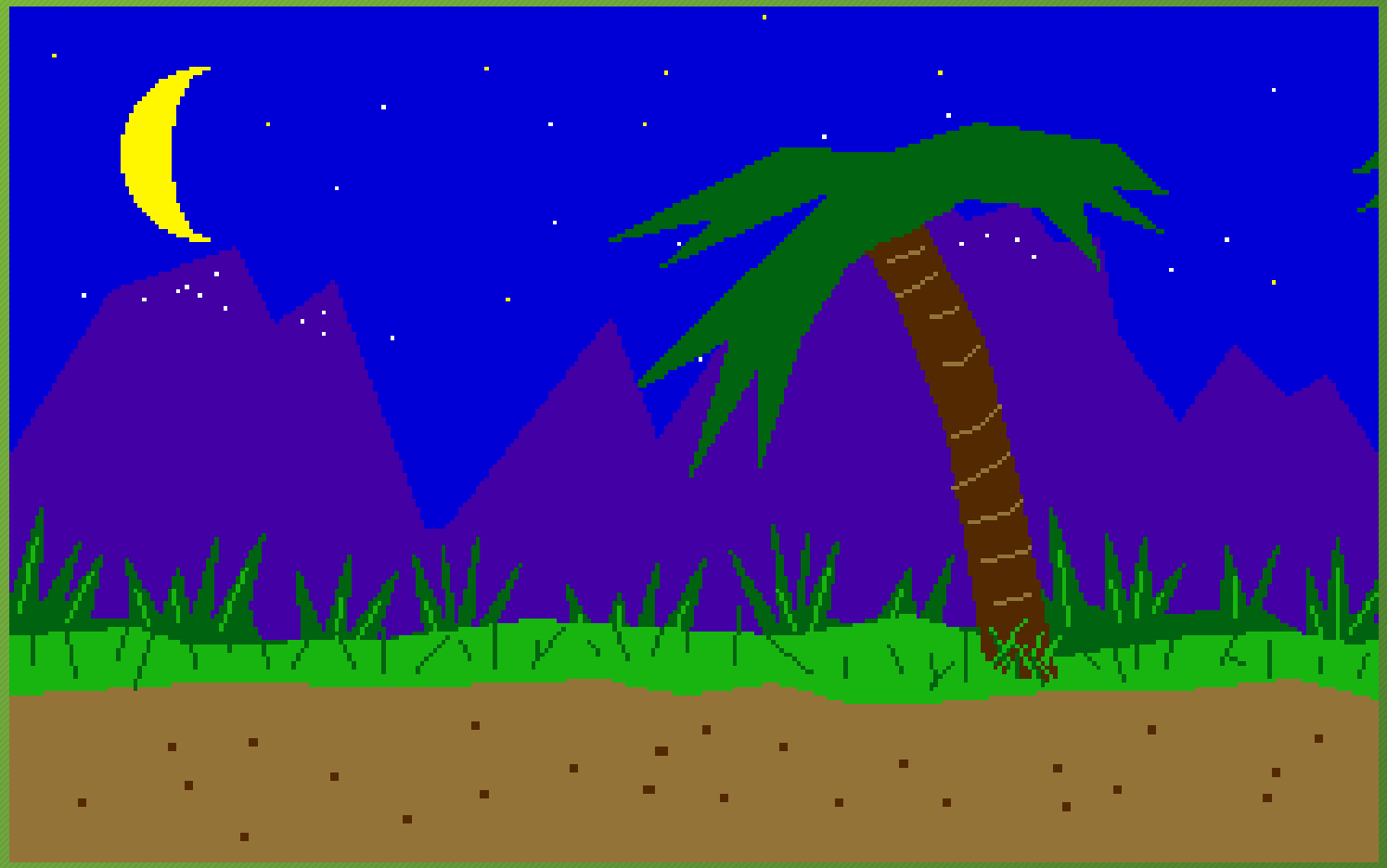
Parallax Scrolling

- Now, imagine that each layer in our tile engine represents one of *x possible depth values: Layer 0 will represent the mountains, layer 1 the* fence, and layer 2 the foreground.
- If we move the different layers at decreasing speeds (depending on their depth), we could somehow fool the eye and achieve a sense of depth similar to that found in 3D viewing.
- This technique is what we usually refer to as parallax scrolling: storing depth-layered tiles and moving them at different speeds to convey a sense of depth.

A general Parallax Rendering Loop

```
if (pressed the right cursor)
    for (layeri=0;layeri<numlayers;layeri++) playerx+=1*(layeri+1);

for (layeri=0;layeri<numlayers;layeri++)
    {
    for (yi=beginy;yi<endy;yi++)
        {
        for (xi=beginx;xi<endx;xi++)
            {
            int screenx=xi*tile_wide – playerx[layeri]-screenplayerx;
            int screeny=yi*tile_high – playery[layeri]-screenplayery;
            int tileid=mapping_matrix [layeri][yi][xi];
            if (tileid>0) blit(tile_table[tileid],screenx,screeny);
        }
    }
```

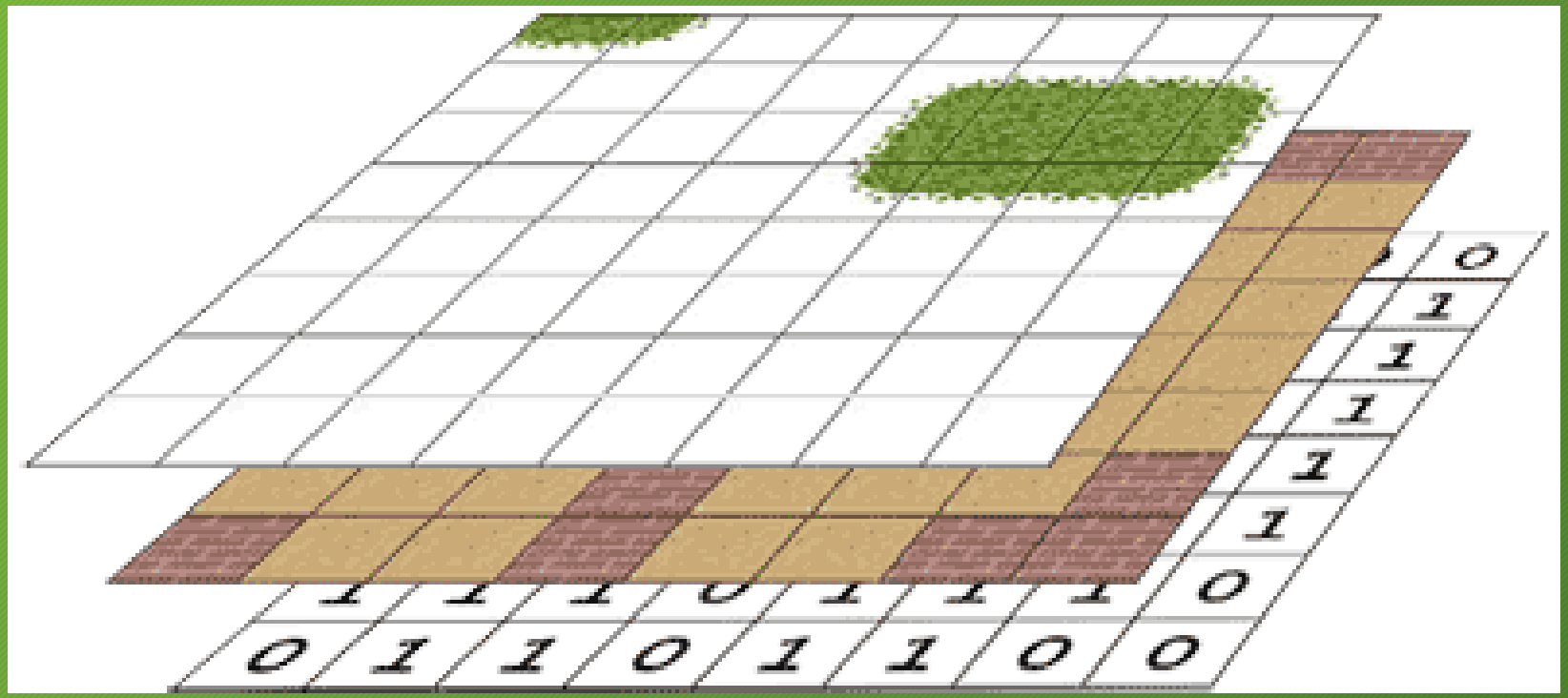
Simple Parallax

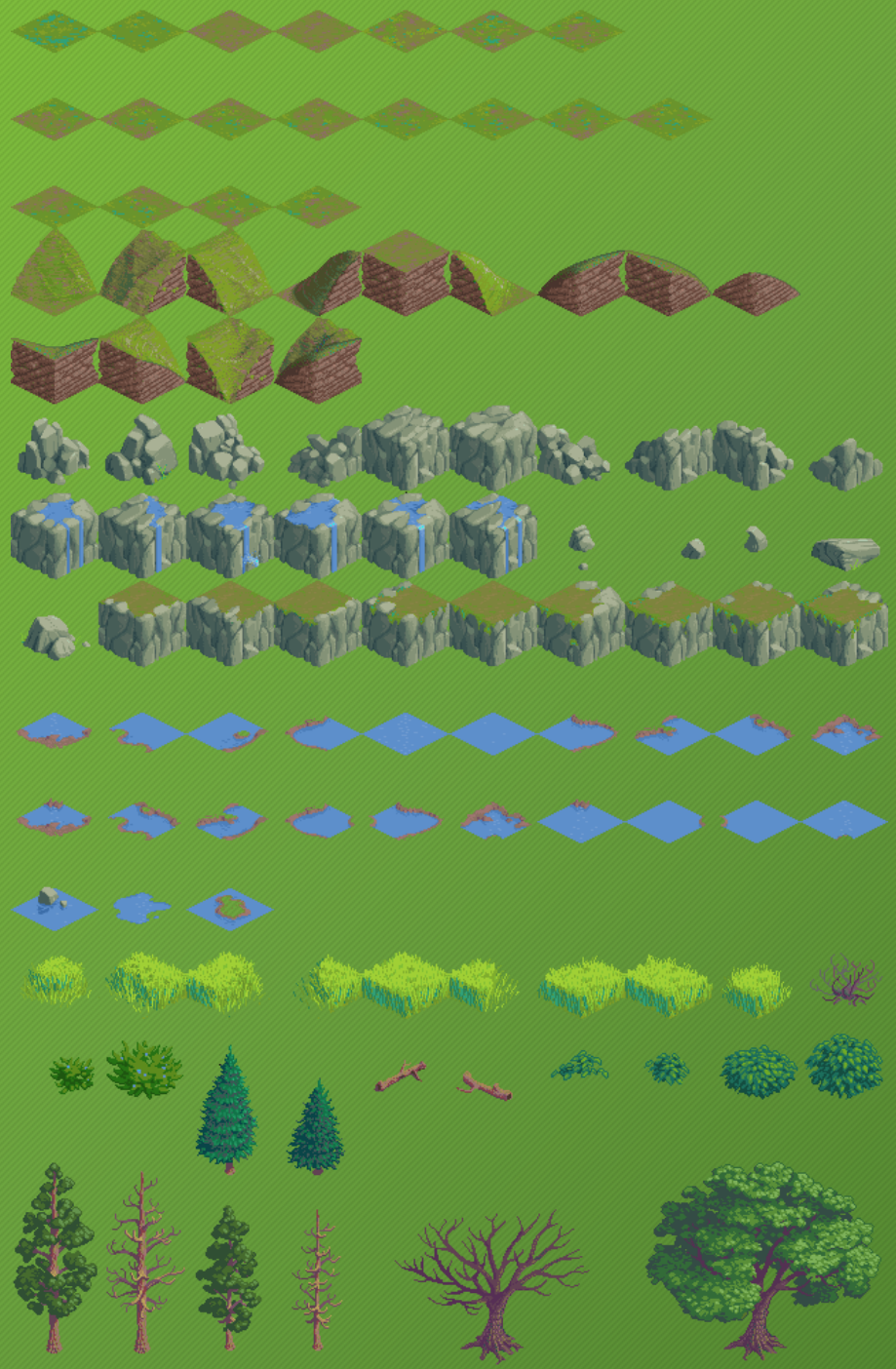


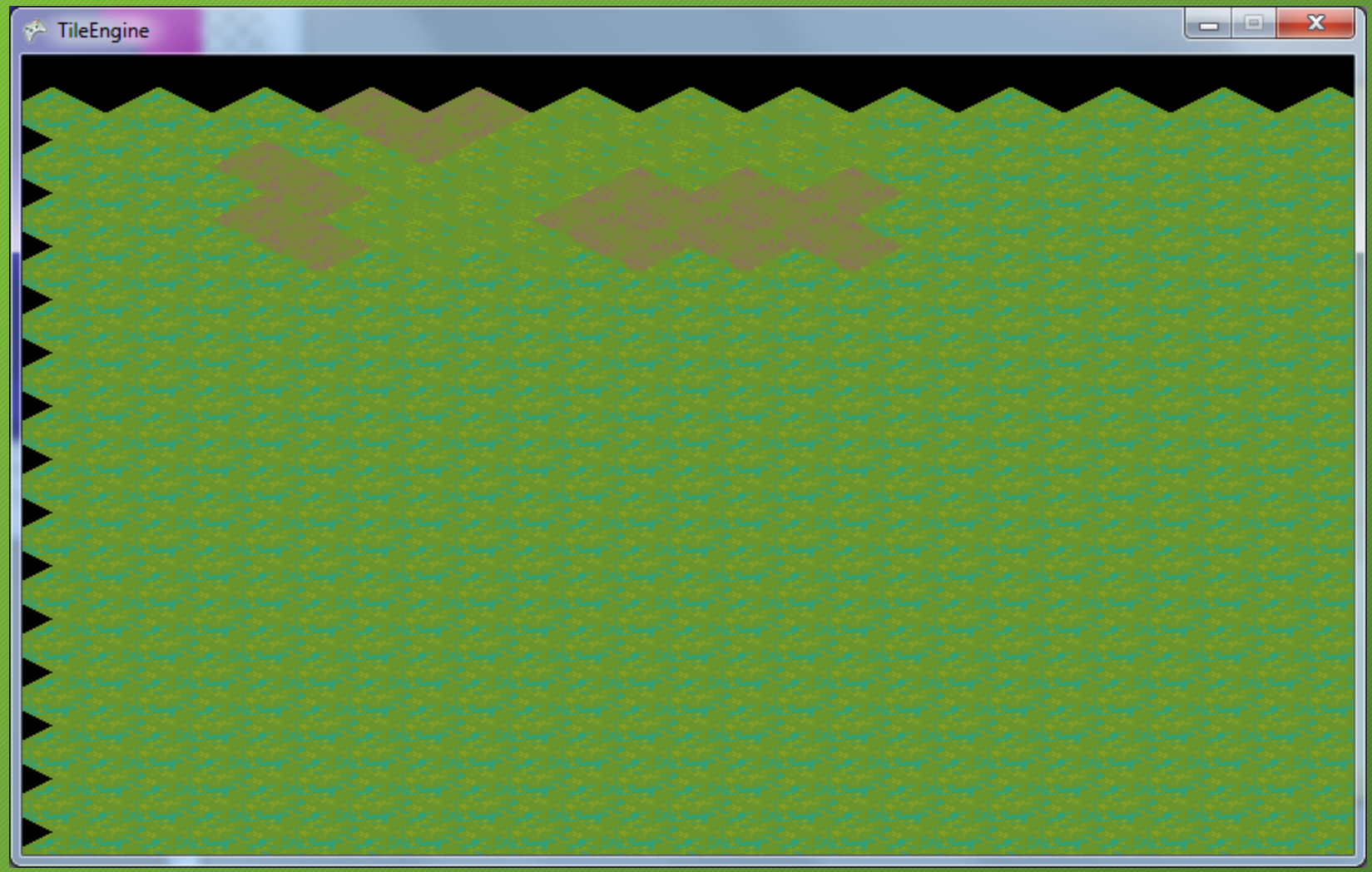
Isometric Engines

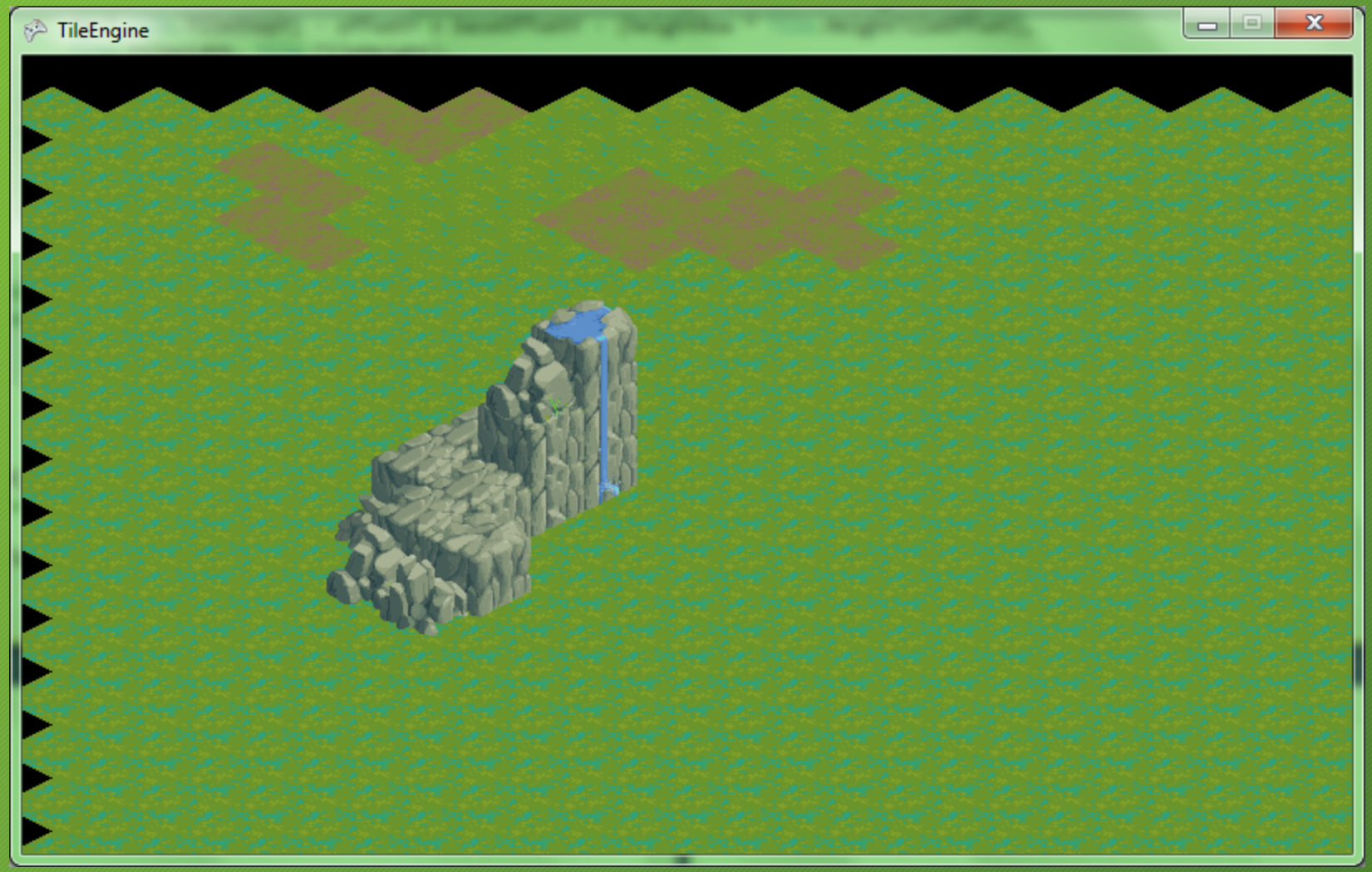
- Isometric perspective consists of representing an object or game level sideways and from a raised viewpoint, as if it was rotated 45°.
- It is a parallel perspective, meaning that lines do not converge as in conic perspective projection.
- Thus, isometric perspective does not suffer from distortion and keeps the object's real proportions and relationships.

Isometric Perspective






TileEngine


TileEngine

Page Swap Scrollers

- Another variant of the classic scrolling algorithm can be used in those games where we want to offer a scrolling playing field without being restricted to a closed set of tiles.
- Imagine a level that offers the game play smoothness of a scroller, but where every single piece of art is indeed unique.
- Games such as *Baldur's Gate or the Monkey Island saga come to mind as classic examples of page-swap scrolling.*

Page Swap Scrollers

- The starting point is to work on the level map as a single image, and then divide it into sectors, much like checkers on a checkerboard.
- Then, we use the player's position to determine which sectors should be loaded into main memory.
- The rest are stored on secondary media (hard drive, CD, etc.), but will be swapped into main memory as needed. It is all a matter of ensuring that our disk bandwidth is good enough so that the player does not notice the loading going on in the background.
- The smaller the rectangle, the less noticeable it will be.
- The mapper thus resembles a cache memory: keeping frequently used data within fast reach and swapping it to ensure that we always maintain optimal performance.

Matrix Shifting

```
class tile *tileptr;

class pagescroller
      {
      tileptr **matrix;
      int sizex,sizey;
      int cornerx,cornery;
};
```

```
for (int iy=0;iy<sizey-1;iy++)
      {
      aux=matrix[iy][0];  // the first element is really thrown away
for (int ix=0;ix<sizex-1;ix++)
      {
            matrix[iy][ix]=matrix[iy][ix+1];
            }
      matrix[iy][sizex-1]=aux;
      Fill matrix[iy][sizex-1] with new data from the sector appearing in high X
      }
```