Collision Detection and Response for Platformers

aie
SPECIALIST EDUCATORS IN
GAMES, ANIMATION & FILM VFX

Topics

- Level data (simplifying collision objects)
- Collision detection on a grid
- Applying forces
- Testing vertically
- Testing horizontally
- The collision algorithm

Simplifying Level Data for Collisions

- We can draw our level by reading the JSON level data
- BUT…
  - The tileset images are 70px x 70px
  - The map tiles are 35px x 35px
  - Testing the character for collisions against every tile in the map is slow

Simplifying Level Data for Collisions

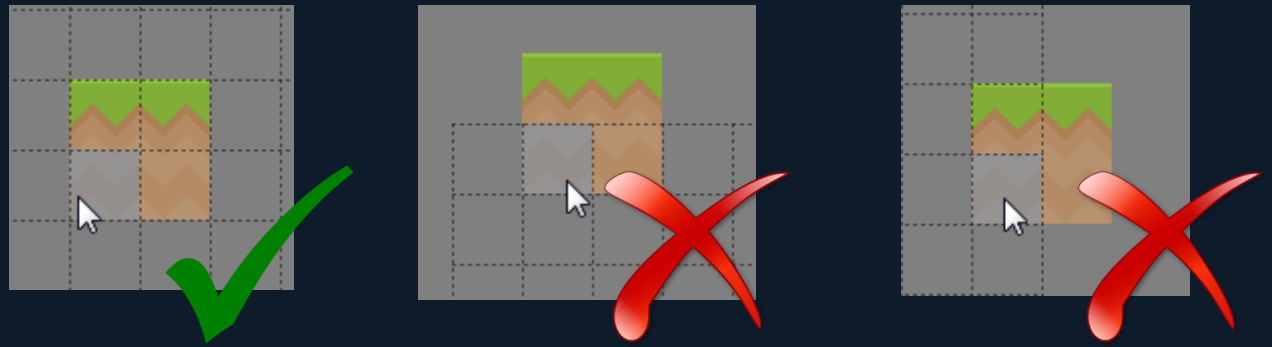
- To simplify, make a grid (2D array) of collision data
- Each array element represents 1 35x35 tile
- 0 = no collision, 1 = collision
- We'll make a function to initialize the game

Simplifying Level Data for Collisions

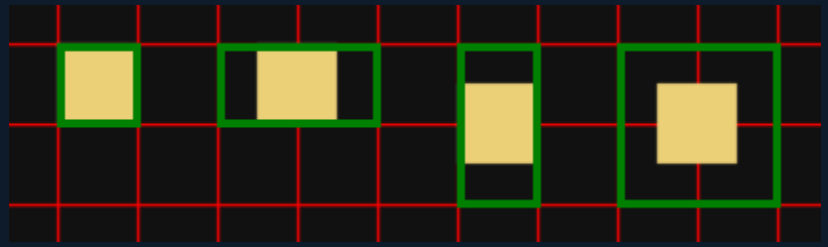
```
var cells = [];                         // the array that holds our simplified collision data
function initialize() {
    for(var layerIdx = 0; layerIdx < LAYER_COUNT; layerIdx++) {    // initialize the collision map
        cells[layerIdx] = [];
        var idx = 0;
        for(var y = 0; y < level1.layers[layerIdx].height; y++) {
            cells[layerIdx][y] = [];
            for(var x = 0; x < level1.layers[layerIdx].width; x++) {
                if(level1.layers[layerIdx].data[idx] != 0) {
                        // for each tile we find in the layer data, we need to create 4 collisions
                        // (because our collision squares are 35x35 but the tile in the level are 70x70)
                    cells[layerIdx][y][x] = 1;
                    cells[layerIdx][y-1][x] = 1;
                    cells[layerIdx][y-1][x+1] = 1;
                    cells[layerIdx][y][x+1] = 1;
                }
                else if(cells[layerIdx][y][x] != 1) {
                    cells[layerIdx][y][x] = 0;     // if we haven't set this cell's value, then set it to 0 now
                }
                idx++;
            }
        }
    }
}
```

Simplifying Level Data for Collisions

- The origin for the 70x70 tile is the bottom left
- This algorithm works if you:
  - Don’t place a tile on the top-most row
  - Don’t place a tile on the right-most column
- We have 1 collision layer for platforms, and 1 for ladders



Collision Detection on a Grid



- The player collision rect is the same size as a tile
  - Only for platform/ladder collisions
  - Greatly simplifies collision detection logic
- We know the player only ever occupies 1, 2 or 4 cells
- I know the player sprite is bigger! We'll use a different algorithm for bullet collisions

Collision Detection on a Grid

- Look at the 1 to 4 cells that the player occupies
- cell / cellright / celldown / celldiag tell us which cell(s) the player occupies

```
var tx          = pixelToTile(player.x),
    ty          = pixelToTile(player.y),
    nx          = player.x%TILE,      // true if player overlaps right
    ny          = player.y%TILE,      // true if player overlaps below
    cell        = cellAtTileCoord(tx, ty),
    cellright   = cellAtTileCoord(tx + 1, ty),
    celldown    = cellAtTileCoord(tx, ty + 1),
    celldiag    = cellAtTileCoord(tx + 1, ty + 1);
```

Collision Detection on a Grid

- So far, our player's update function looks like this:
- Debug the code, confirm the values for cell, cellright, celldown & celldiag
- Will be 1 if player is in that cell

```
Player.prototype.update = function(deltaTime)
{
   // calculate the new position and velocity:
   this.position.y = Math.floor(this.position.y  + (deltaTime * this.velocity.y));
   this.position.x = Math.floor(this.position.x  + (deltaTime * this.velocity.x));

   // collision detection
   // Our collision detection logic is greatly simplified by the fact that the player is a rectangle
   // and is exactly the same size as a single tile. So we know that the player can only ever
   // occupy 1, 2 or 4 cells.

   // This means we can short-circuit and avoid building a general purpose collision detection
   // engine by simply looking at the 1 to 4 cells that the player occupies:
    var tx = pixelToTile(this.position.x);
    var ty = pixelToTile(this.position.y);
    var nx = (this.position.x)%TILE;        // true if player overlaps right
    var ny = (this.position.y)%TILE;        // true if player overlaps below
    var cell = cellAtTileCoord(LAYER_PLATFORMS, tx, ty);
    var cellright = cellAtTileCoord(LAYER_PLATFORMS, tx + 1, ty);
    var celldown = cellAtTileCoord(LAYER_PLATFORMS, tx, ty + 1);
    var celldiag = cellAtTileCoord(LAYER_PLATFORMS, tx + 1, ty + 1);
}
```

Applying Forces

- Constant variables control the forces in our game

```
    // abitrary choice for 1m
var METER = TILE;
    // very exaggerated gravity (6x)
var GRAVITY = METER * 9.8 * 6;
    // max horizontal speed (10 tiles per second)
var MAXDX = METER * 10;
    // max vertical speed   (15 tiles per second)
var MAXDY = METER * 15;
    // horizontal acceleration -  take 1/2 second to reach maxdx
var ACCEL = MAXDX * 2;
    // horizontal friction     -  take 1/6 second to stop from maxdx
var FRICTION = MAXDX * 6;
    // (a large) instantaneous jump impulse
var JUMP = METER * 1500;
```

Applying Forces

- Check for keyboard left / right / jump
  - Slow down using friction by checking if player was moving left/right
- Apply gravity
- If dx (delta x) is the change in x (velocity), ddx is the change in velocity (acceleration)

Applying Forces

```
if (left)
    ddx = ddx - ACCEL;                  // player wants to go left
  else if (wasleft)
    ddx = ddx + FRICTION;               // player was going left, but not any more

  if (right)
    ddx = ddx + ACCEL;                  // player wants to go right
  else if (wasright)
    ddx = ddx - FRICTION;               // player was going right, but not any more

  if (jump && !this.jumping && !falling)
  {
    ddy = ddy - JUMP;                   // apply an instantaneous (large) vertical impulse
    this.jumping = true;
  }

  // calculate the new position and velocity:
  this.position.y = Math.floor(this.position.y  + (deltaTime * this.velocity.y));
  this.position.x = Math.floor(this.position.x  + (deltaTime * this.velocity.x));
  this.velocity.x = bound(this.velocity.x + (deltaTime * ddx), -MAXDX, MAXDX);
  this.velocity.y = bound(this.velocity.y + (deltaTime * ddy), -MAXDY, MAXDY);
```

Gettin’ Jiggy wit’ it

- Once we start testing collisions (and the player stops falling off the screen) stopping after movement will produce jiggle
- When stopping, the frictional force applied is highly unlikely to be *exactly* the force needed to stop
- The player then starts moving in the opposite direction (repeatedly), producing jiggle

```
if ((wasleft  && (this.velocity.x > 0)) || (wasright && (this.velocity.x < 0)))  {
   // clamp at zero to prevent friction from making us jiggle side to side
   this.velocity.x = 0;
}
```

Testing Vertically

- If player is moving vertically:
  - Check if they have hit a platform below or above
  - If so, stop vertical velocity, clamp Y position

Testing Vertically

```
// If the player has vertical velocity, then check to see if they have hit a platform below
// or above, in which case, stop their vertical velocity, and clamp their y position:
if (this.velocity.y > 0)  {
   if ((celldown && !cell) || (celldiag && !cellright && nx))  {
      this.position.y = tileToPixel(ty);          // clamp the y position to avoid falling into platform below
      this.velocity.y = 0;                        // stop downward velocity
      this.falling = false;                       // no longer falling
      this.jumping = false;                       // (or jumping)
      ny = 0;                                     // no longer overlaps the cells below
   }
}
else if (this.velocity.y < 0)  {
   if ((cell && !celldown) || (cellright && !celldiag && nx))  {
      this.position.y = tileToPixel(ty + 1);      // clamp the y position to avoid jumping into platform above
      this.velocity.y = 0;                        // stop upward velocity
      cell = celldown;                            // player is no longer really in that cell, we clamped them to the cell below
      cellright = celldiag;                       // (ditto)
      ny = 0;                                     // player no longer overlaps the cells below
   }
}
```

Testing Horizontally

- Apply similar logic for testing horizontally
- cellAtTileCoord returns 1 (collision) if pixel coordinates are off-screen

```
if (this.velocity.x > 0) {
   if ((cellright && !cell) || (celldiag  && !celldown && ny))  {
      this.position.x = tileToPixel(tx);              // clamp the x position to avoid moving into the platform we just hit
      this.velocity.x = 0;                            // stop horizontal velocity
   }
}
else if (this.velocity.x < 0) {
   if ((cell && !cellright) || (celldown && !celldiag && ny))  {
      this.position.x = tileToPixel(tx + 1);          // clamp the x position to avoid moving into the platform we just hit
      this.velocity.x = 0;                            // stop horizontal velocity
   }
}
```

And Finally…

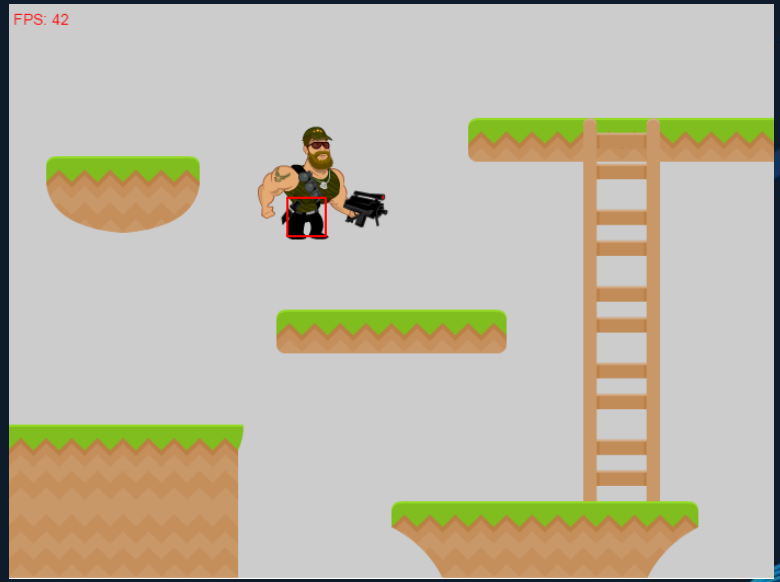
- Lastly, check if the player is falling or not
- Look to see if there is a platform below

```
player.falling   =   ! (celldown || (nx && celldiag));
```

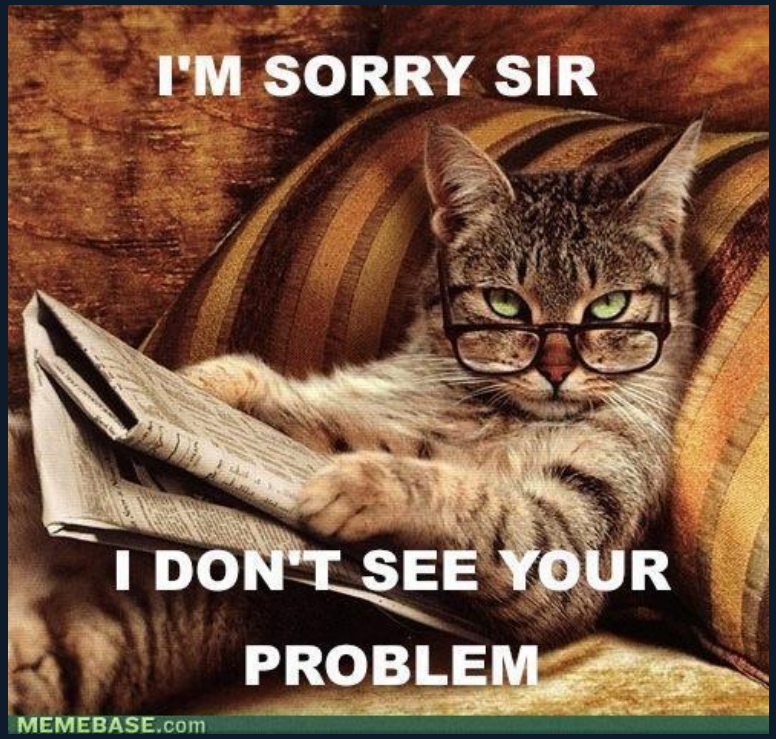
- (This means we can’t jump after walking off a cliff)

Summary

- You should now have a functioning platformer (run & jump)
- To climb, test if player near ladder and pressing up/down
- Climbing can be a different player state



Questions



aie
SPECIALIST EDUCATORS IN GAMES, ANIMATION & FILM VFX