Peter Mileff PhD

Programming of Graphics

Introduction to 2D graphics

University of Miskolc
Department of Information Technology

Game scenes, object, management…

What else is needed?

- GameObject is only capable of storing animations

A game is a complex system

- **A global management is needed for:**
  - the objects: a game is a big set of (moving) objects
  - For the world:
    - Need a complex structure for handling scenes, layers and the whole world

The 2D world

- A “complex” game requires independently handled scenes
  - For example, in a 2D game we go in through a door and another part opens in front of us
- So we can model the 2D world, the game, with a set of scenes

- What is needed for the implementation:
  - A unit capable of storing a scene is required
  - We need a manager that brings them together

Jelenet menedzser

- The task of the manager is to combine the scenes into one unit
  - switching, adding, deleting scenes, etc.

```
class C2DSceneManager {
   vector<C2DScene> mScenes;
…
public:

void AddScene(CScene scene);
void RemoveScene(int id);
CScene GetScene(int id);
...
};
```

The Scene class

- The objective of the scene class (C2DScene) is to make a logical unit for all the (game) objects in the scene

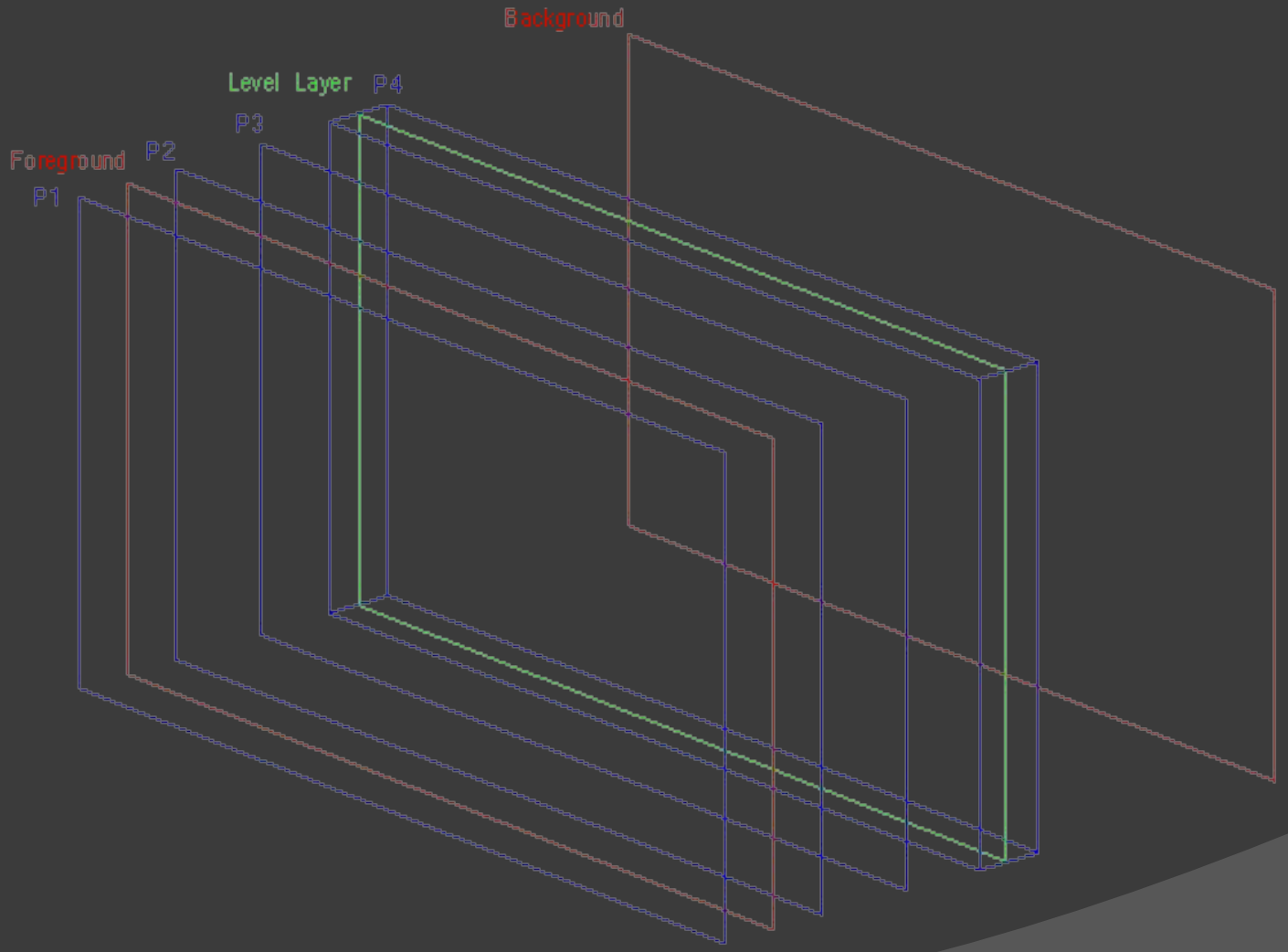
```
class C2DScene {
    /// Layer for objects
    CVector<C2DGraphicsLayer> mLayers;

    /// Name of the scene
    string mName;

    /// Visibility flag
    bool mVisible;

    /// Global camera for the whole scene
    CCamera2D mSceneCamera;

public:
…
};
```

The Scene class

- A 2D world is preferably made up of different **layers**
  - makes implementation easier
  - E.g.: 2D side scroller game
    - 1 layer: background
    - 2 layer: where the real game logic is happening
    - 3 layer: other moving elements in the foreground
- Other important attributes:
  - The scene should be named, because it is easier to refer to
  - Turn visibility on and of
    - some parts of the game can be temporarily turned off
  - Global scene camera
    - For scrolling the level (map)

Example

Example



The C2DGRaphicsLayer class

- It combines a specific level

```
class C2DGraphicsLayer {
 vector<CGameObject2D> mObjectList; // Game Object List
 vector<CTexture2D> mTextures;      // Background and static textures
 bool mVisible;                     // Is layer visible or not
 bool mEnableCollision;             // Enable Collision on Layer
 CCamera2D mCamera;              // Camera for the layer
 CString mName;                     // Layer name
 int mID;                           // Layer ID
 C2DScene mParentScene;             // Parent Scene

public:

...
};
```

The C2DGRaphicsLayer class

- **THe most important data:**
  - The list of game objects
  - The list of static, simple textures
    - Simple textures are often needed. E.g.: background
  - Turn on/off visibility
  - Layer camera
    - Scrolling the screen, the layer
  - The name and ID of the layer
    - For easier refer to the layer
  - Hold a reference to its parent
    - Only for practical reason

Storing the game world...

Storing the game world

- The game world must be stored persistently somewhere
- The best solution is the **binary** format
  - protects copyrights
  - allows fast loading
- You should also use a text-based solution for development:
  - JSON
  - XML
  - other

An XML based example

```
<?xml version="1.0" encoding="utf-8"?>
<scene name="Platformer Demo Scene" layers="4" >
  <layer id="0" name="Sky layer">
    <texture id="0" x="0" y="0" file="2DScene/sky.pcx" />
  </layer>
  <layer id="1" name="Mountain Layer">
    <texture id="1" x="0" y="0" file="2DScene/mountain.tga" />
   </layer>
  <layer id="2" name="Ground Layer">
    <texture id="2" x="0" y="0" file="2DScene/ground.tga" />
  </layer>
  <layer id="3" name="Character layer">
      <gameobject id="777" name="Liza" collidable="1" zindex="0">
      <sprite file="gamedemo/girl/girl.ani" />
      <position x="450" y="565" />
      <direction x="1" y="0" />
      <scale x="1" y="1" />
      <speed value="0" />
      <rotate value="0" />
    </gameobject>
   </layer>
</scene>
```

Bounding Object based rendering…

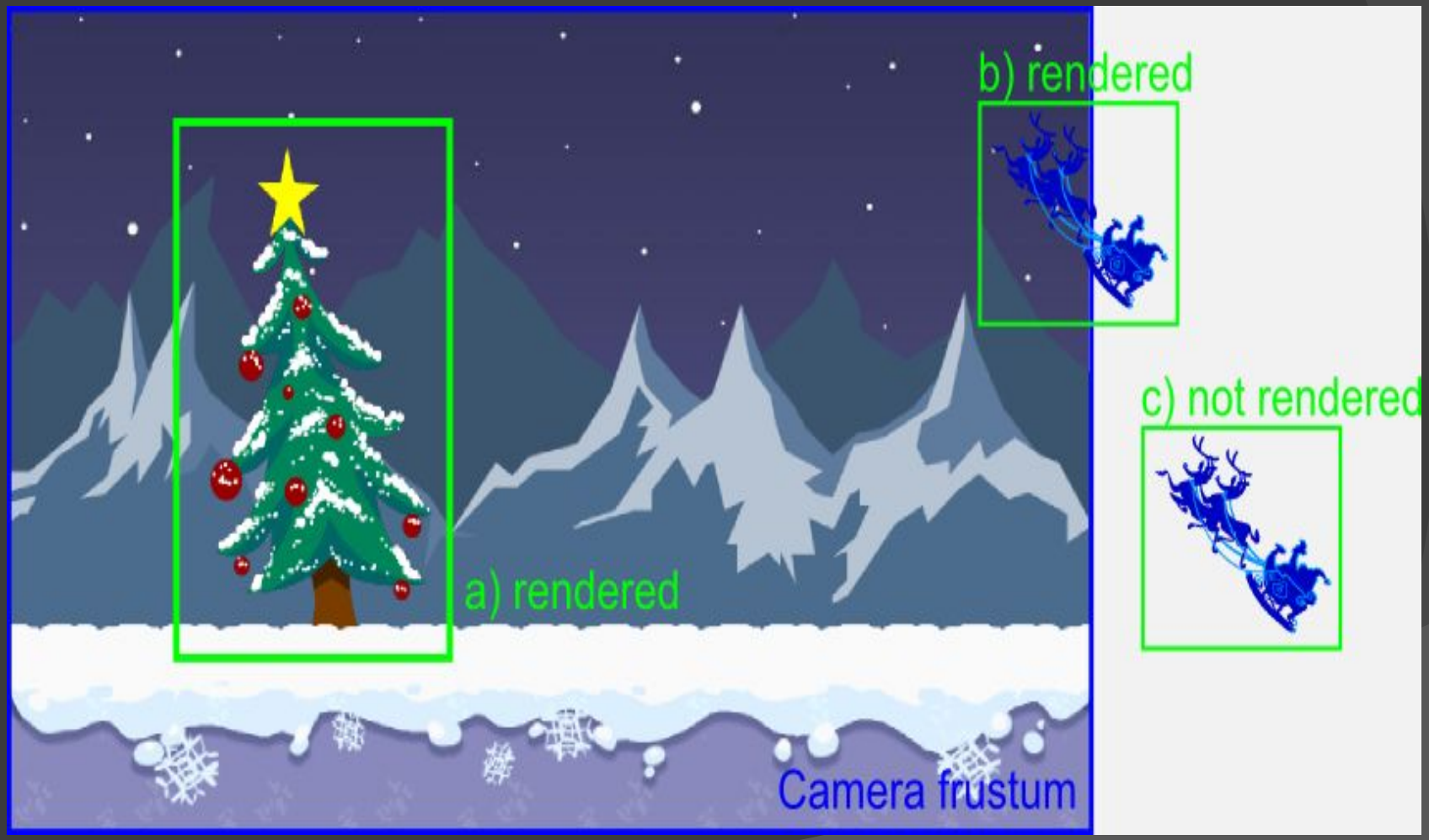
Bounding object based rendering

- The solution is used in most two-dimensional (and 3D) games
  - Reduces drawing data using bounding objects
    - To speed up algorithms
- What is the general logic behind it?
  - It is advisable to handle any visualization task in larger units
    - E.g.: drawing, collision detection, etc
  - The bigger is a unit, the bigger performance we reach

Bounding object based rendering

- **Bounding object technique:**
  - A minimal bounding entity should be determined (or set) for every object
    - in the simplest case, it is a rectangle or a circle,
    - in more complex cases, it is given by polygons
  - In practice the most common used entity in the **Bounding Box**
- **The logic of the algorithm:**
  - When displaying objects:
    - before each drawing frame, we check that the bounding box of that object is in the area of the screen
    - 2D Camera frustum: {0-width, 0-height}

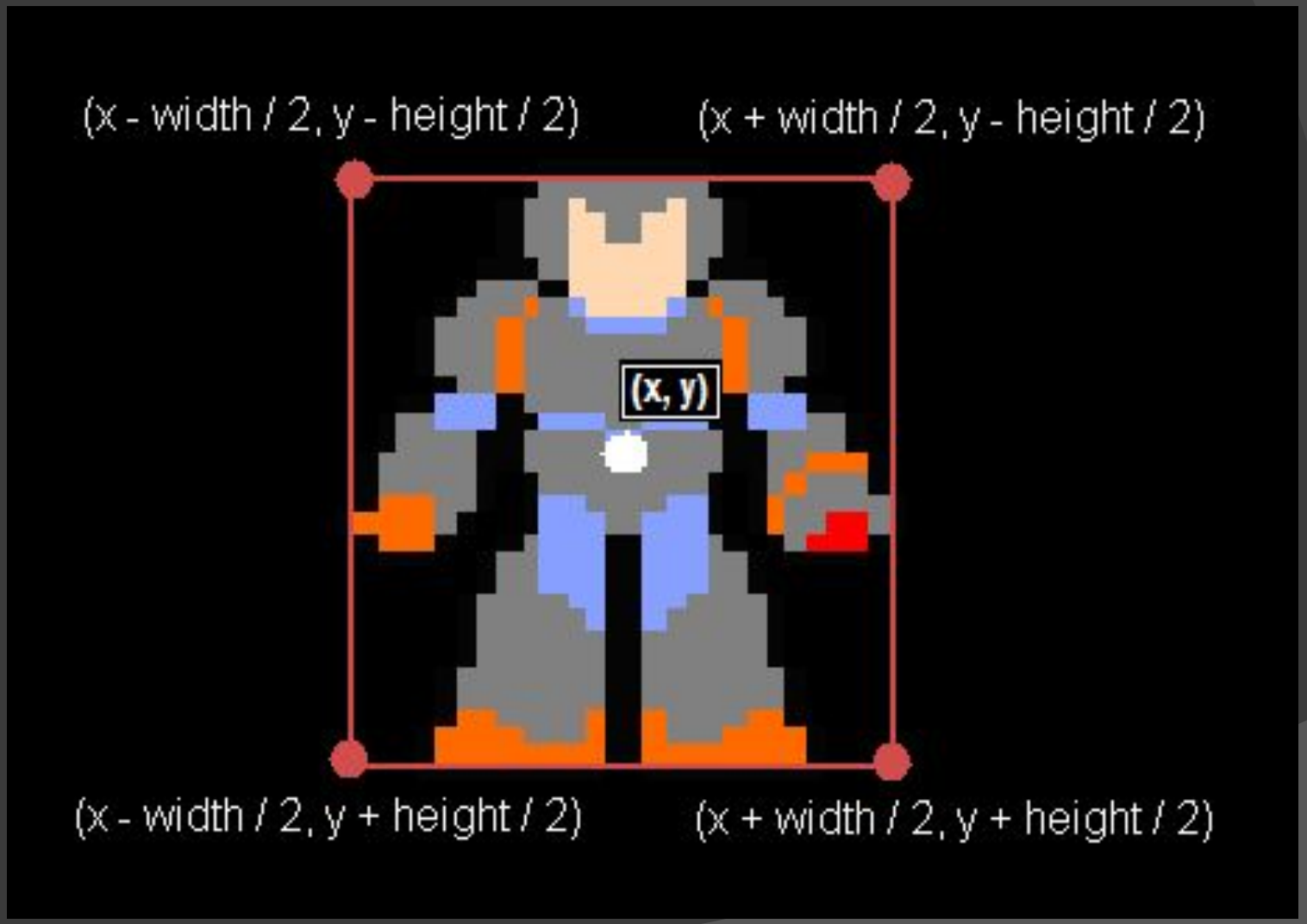
Bounding object based rendering



Bounding object based rendering

- **Bounding box features:**
  - Box should be an exact matching sized shape
    - Usually defined when loading Sprite
  - **Reason:** To avoid calculation errors. Example:
    - BB contains some transparent pixels on the right
    - The object is positioned so that only these pixels are inside the screen
    - This will cause the object to be rendered, even though nothing is visible
    - It goes through the graphical API pipeline, transformed, uses resources

Minimal Bounding Box



A doboz általában megegyezik a kép szélességével és magasságával

Bounding object based rendering

Why box or circle?

- Because they are very simple elements
- Later calculations with them:
  - simple, low CPU resource required
    - E.g.: collision detection, rotation, scale, translate, etc
- Although they do not cover the object well, they are effective and well applied in practice

Sample BB class

```
/// 2D Axis Aligned Bounding Box
class CBoundingBox2D {
    Vector2 minpoint;                  // Box minpoint
    Vector2 maxpoint;                  // Box maxpoint
    Vector2 bbPoints[4];               // bounding box points
    float boxHalfWidth;                   // box half width
    float boxHalfHeight;             // box half height
    matrix4x4f tMatrix;              // Transformation matrix
 public:
...
};
```

Example BB class features

- In a two-dimension, a bound box can be defined by 4 points,
  - 4 corners of the box
- In order to speed up later calculations:
  - it is advisable to store the minimum and maximum points of the screen relative to the coordinate system
    - this usually means the upper left and lower right points
  - It is advisable to store half width and half height values as well

Example BB class features

- When moving an object (shifting, rotating, stretching), the box coordinate must also be transformed
  - The matrix member variable in the class can help with this
- **Time of calculation:**
  - As the object moves, we calculate the new position/orientation of th BB as well
    - Calculation need to be done for every object in every frame

BB based rendering

To decide if an object can be displayed or not:

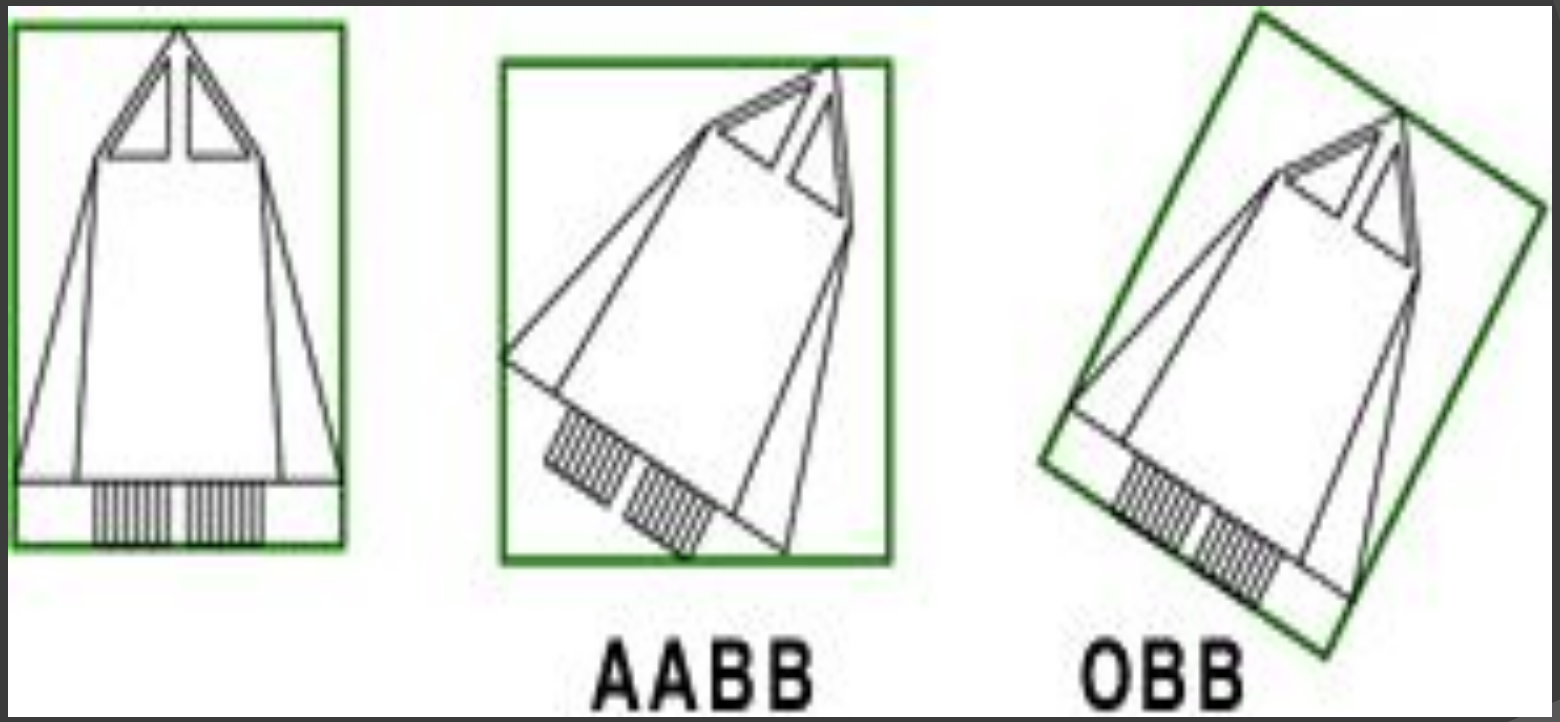
```
if (bb.maxpoint.x < 0 || bb.minpoint.x > screen_width ||
    bb.minpoint.y > screen_height || bb.maxpoint.y < 0)
{
        return false;
}
```

Rotating the Bounding Box…

Group of Bounding Boxes

- The box should be rotated when the object is rotating
  - The corners must be transformed manually
- **Based on this, there are two groups:**
- **Axis-Aligned Bounding Boxes (AABB):**
  - a rectangle whose edges are parallel to a coordinate axis
- **Oriented Bounding Box (OBB):**
  - a rectangle that rotates with the object being rotated

AABB and OBB



Bounding Box in practice

- In practice, implementing AABB is much simpler than implementing OBB
- **Significantly easier for AABB:**
  - Collision detection,
  - calculation of overlap between boxes,
  - Cut/drop objects, which are not in the screen rectangle,
  - other calculations
- **Drawback:**
  - the corner points of the box must be recalculated in every rotation
  - This results reduced accuracy at collision detection

Bounding Box in practice

- Recalculate AABB points in three steps:
  - in rotation, we transform the corners of the box,
  - we search for the minimum and maximum points,
  - then we create the new box based on these points
- For OBB:
  - from the above points, it is enough to perform only the first step
  - The difficulty lies in determining whether two boxes overlap
    - Mathematically more complex
- The used solution always depends on the needs of the software being developed
  - In most cases, the AABB more than enough

Rotating the Box (AABB)

- **The algorithm:**
  - Rotate the four corners of the box
  - After rotation, the box is recalculated again

```
for (int i = 0; i < 4; i++) {
    Vector2 point(bbPoints[i].x,bbPoints[i].y);
    m_mTransformationMatrix.rotate_x(point, angle);
    bbPoints[i].x = point.x;
    bbPoints[i].y = point.y;
}
searchMinMax();      // Search min and max points
setUpBBPoints();     // setup AABB box
```

- A rotate_x function rotates the points by the x axis

Rotating the Box (AABB)

```
void searchMinMax() {
    Vector2 min = Vector2(bbPoints[0].x, bbPoints[0].y);
    Vector2 max = Vector2(bbPoints[0].x, bbPoints[0].y);

     for (int i = 0; i < 4; i++) // loop the 4 points
      {
         if (bbPoints[i].x < min.x){
            min.x = bbPoints[i].x;
          }

         if (bbPoints[i].y < min.y){
            min.y = bbPoints[i].y;
        }
```

Rotating the Box (AABB)

```
        if (bbPoints[i].x > max.x){
            max.x = bbPoints[i].x;
        }
        if (bbPoints[i].y > max.y){
            max.y = bbPoints[i].y;
        }
    }
    minpoint = min;
    maxpoint = max;
}
```

Rotating the Box (AABB)

```
void setUpBBPoints(){
    // 1. point
    bbPoints[0].x = minpoint.x;
    bbPoints[0].y = minpoint.y;
     // 2. point
    bbPoints[1].x = maxpoint.x;
    bbPoints[1].y = minpoint.y;
     // 3. point
    bbPoints[2].x = maxpoint.x;
    bbPoints[2].y = maxpoint.y;
     // 4. point
    bbPoints[3].x = minpoint.x;
    bbPoints[3].y = maxpoint.y;
}
```

2D Collision detection…

Ütközésvizsgálat

- An essential element of a game is the interaction of objects
  - The event when two objects collide with each other
- Collision is not only the game feature:
  - the same principles are used when, for example, moving the mouse over a menu item
- Of course, this plays a dominant role in computer games
  - the game experience is highly affected by these interactions
    - For example, in a action game, the projectile hits the enemy

2D collision detection in practice

The essence of the collision detection:

- algorithmically determine that two-dimensional images of two or more objects overlap
- **The exact collision detection:** means that one pixel of an object overlaps the pixel of another object
  - This detection is compute-intensive!
- **Game developers often make fake (not exact) collisions:**
  - try to include the moved elements in some object
    - Bounding box, circle, sphere, polygon, etc
  - Performing the collision test on these objects can reduce the computational demand

2D collision detection in a game engine

- **Collision detection time:**
  - **Before** the moving the objects and the bounding box into the the new position
    - Otherwise, the object stuck in the "wall" and may stick together
  - Each object must be checked with each object

- **The algorithm:**
  - If movement is needed, the new position of the object and the box should be calculated
  - Collision test should be performed using these values
  - If it doesn't collide, the object can have this new position,
  - otherwise we have to decide what to do with the object
    - eg. stops, explodes, etc.

2D collision detection samplep

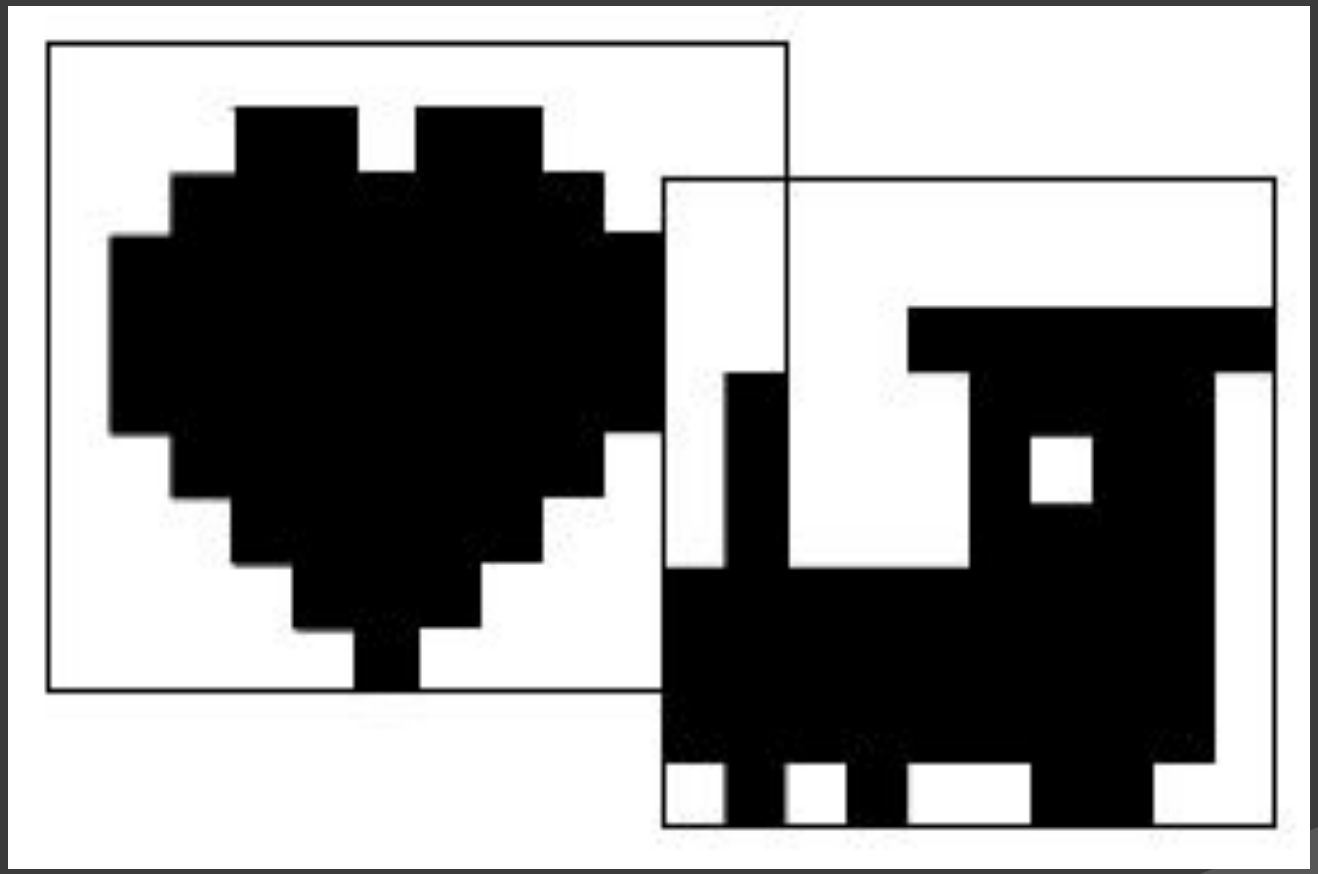
```
void checkCollisions() {
    // check other sprite's collisions
    spriteManager.resetCollisionsToCheck();
    // check each sprite against other sprite objects.
    for (Sprite spriteA : spriteManager.getCollisionsToCheck()) {
        for (Sprite spriteB : spriteManager.getAllSprites()) {
            if (handleCollision(spriteA, spriteB)) {
                // The break helps optimize the collisions
                //  The break statement means one object only hits another
                break;
            }
        }
    }
```

BEFOGLALÓ DOBOZ ALAPÚ ÜTKÖZÉSVIZSGÁLAT…

Befoglaló doboz alapú ütközésvizsgálat

- Az évek során több különféle ütközésvizsgálati technika alakult ki:
  - pl. Separate Axis Theorem
- Legnépszerűbb és egyszerűbb a befoglaló doboz alapú technika
  - Általánosan: rectangular collision detection
- **Lényege:**
  - Az elv azonos az objektumok képernyőn való megjelenítésének vizsgálatával.
  - Amikor két objektum befoglaló doboza (vagy esetleg köre) átfedi egymást, az objektumok ütköznek.

Befoglaló doboz alapú ütközésvizsgálat



- A befoglaló dobozok átfedéséből egyértelműen meghatározható az ütközés ténye.

Egy lehetséges algoritmus

- A gyors ellenőrzés miatt célszerű azt érzékelni mikor nincs ütközés
  - így felesleges számításoktól kíméljük meg a CPU-t

```
bool checkCollision(CBoundingBox2D* boxObj1, CBoundingBox2D* boxObj2){

if (boxObj1->GetMaxPoint()->x < boxObj2->GetMinPoint()->x ||
boxObj1->GetMinPoint()->x > boxObj2->GetMaxPoint()->x){
      // Nincs ütközés
      return false;
 }
 if (boxObj1->GetMaxPoint()->y < boxObj2->GetMinPoint()->y ||
boxObj1->GetMinPoint()->y > boxObj2->GetMaxPoint()->y){
          //nincs utkozes
          return false;
 }
    return true;
 }
```

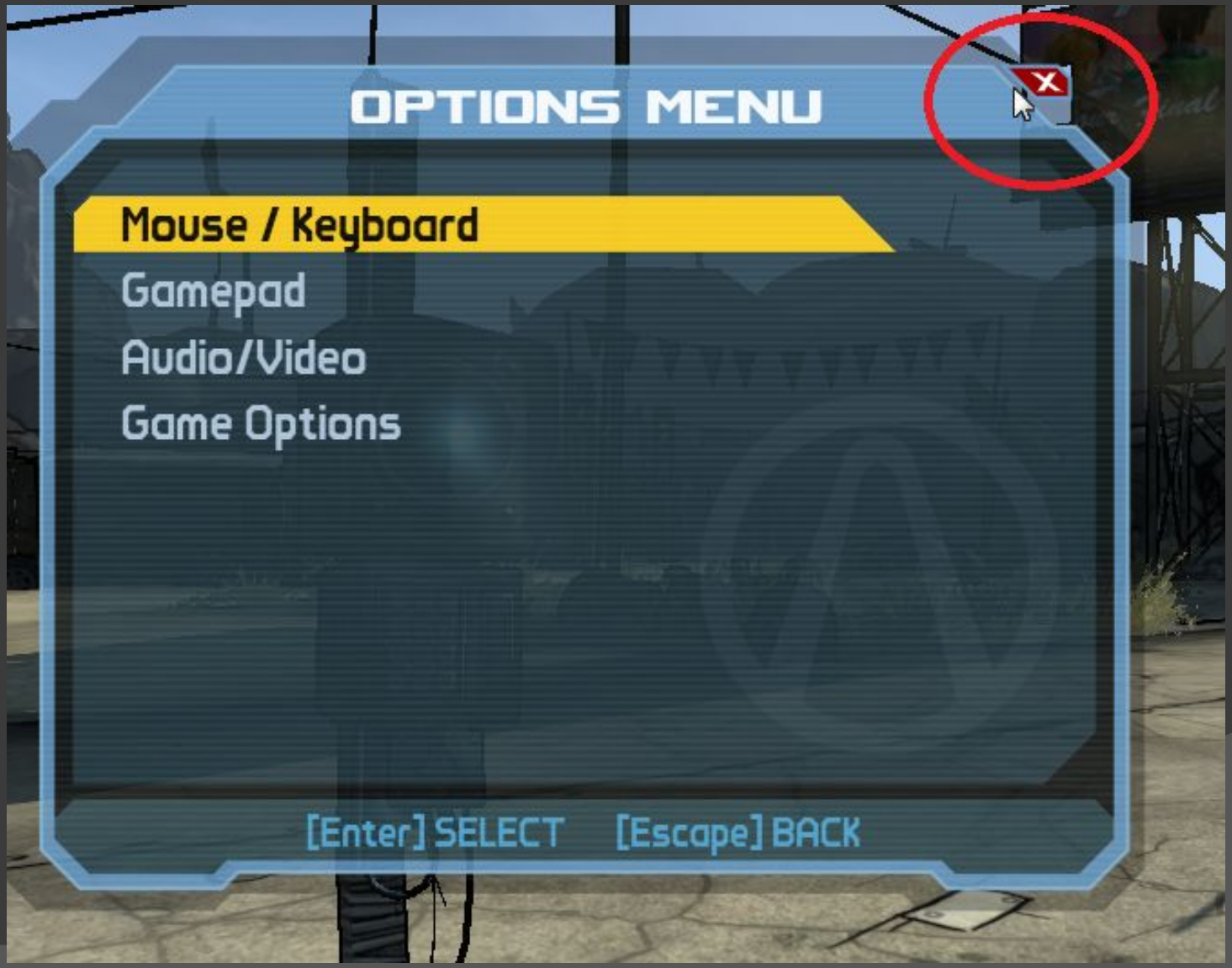
Befoglaló doboz alapú megoldás hibái

- **„Lyukas” objektumok ütközése:**
  - objektumok átlátszó résszel
  - Ha valójában a lyukas részek fedik csak át egymást, úgy nem történik tényleges ütközés!
- A hiba ellenére a játékfejlesztés területén ez a megoldás terjedt el leginkább.
- **Oka:**
  - az egyszerűsége és a redukált számításigény
  - a legtöbb játék esetében gyors mozgás közben nem vesszük észre, hogy „nem is az objektum pixelével ütköztünk”.

Példa ismert „hibákra”

- A mai modern játékok menürendszere
  - pl. BorderLands - Unreal Engine, Crysis sorozat, stb
  - Szinte minden játék esetében
- **Érzékelés:**
  - Egy nem szabályos, például rombusz alakú gomb kiválasztása
  - Az alsó, nem valós területére mozdítva az egeret a felhasználói interakció megtörténik (a gomb kivilágít).

Borderlands - Menü



A hiba orvosolása

- Kialakult egy nagyon egyszerű, de könnyen megvalósítható megoldás a gyakorlatban.
- A befoglaló doboz méretét nem pixelre pontosan az objektum képére számolják ki
- Redukálják annak méretét valamilyen értékkel.

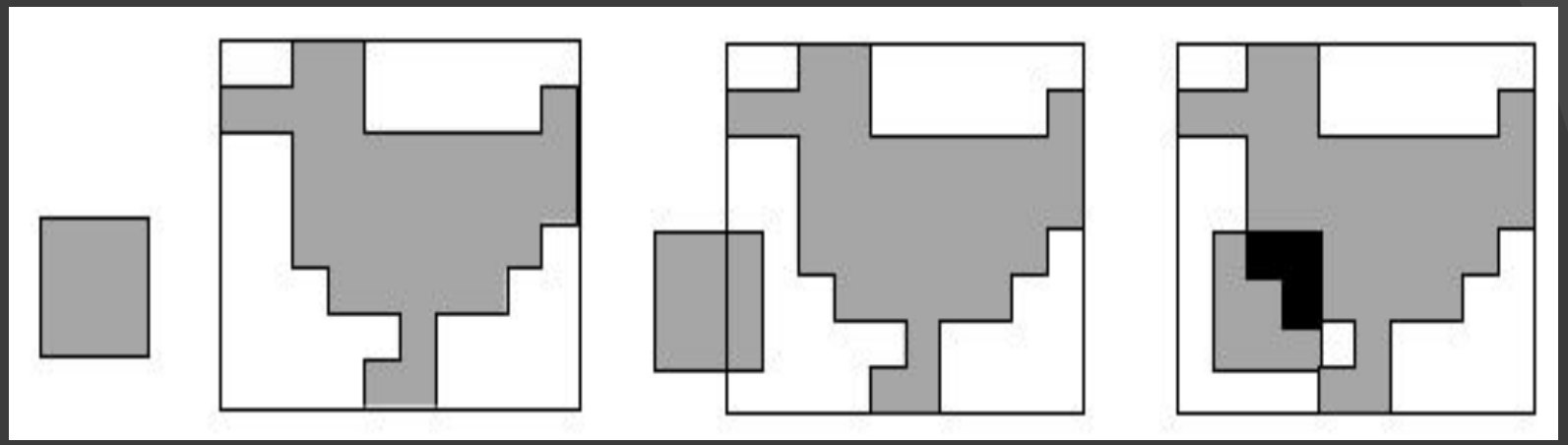
```
object->width = <ACTUAL BITMAP WIDTH>;
object->height = <ACTUAL BITMAP HEIGHT>;
object->col_width = object->width * 0.80;
object->col_height = object->height * 0.80;
object->col_x_offset = (object->width - object->col_width) / 2;
object->col_y_offset = (object->height - object->col_height) / 2;
```

PIXEL SZINTŰ ÜTKÖZÉSVIZSGÁLAT…

Pixel szintű ütközésvizsgálat

- **Elnevezés:** **per-pixel collision detection**
- Valódi, pontos ütközésvizsgálat
- Minden szoftver esetében a pixel alapú megoldás lenne az ideális
- Számításigénye nagy!
- Emiatt azonban csak ott alkalmazzák, ahol erre kimondottan igény van.

Pixel szintű ütközésvizsgálat



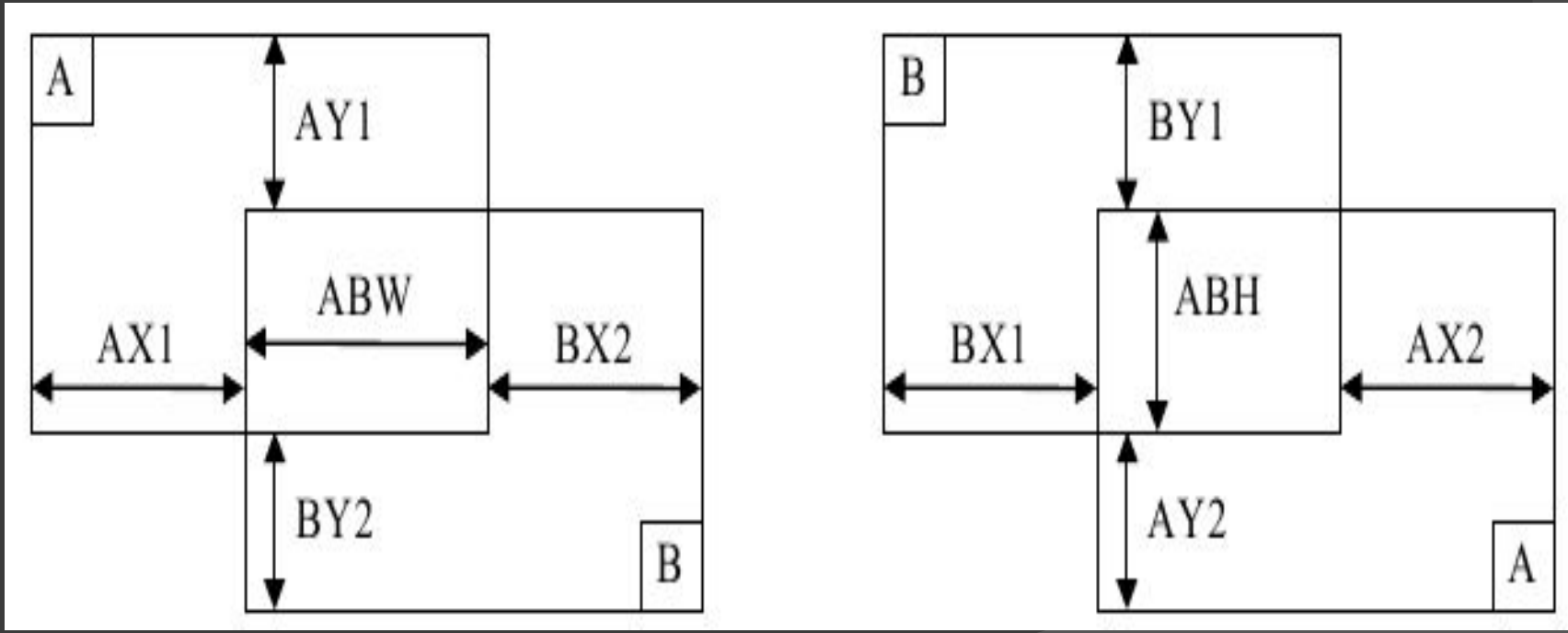
- **Bal oldal:** nincs ütközés
- **Jobb oldal:** valós ütközés
  - Mivel a pixelek takarják egymást, a befoglaló dobozok is
- **<u>Középső:</u>**
  - a „golyó” még nem hatolt be a kacsa „testébe”, viszont a mintáját tartalmazó téglalap alakú tartományba már igen
  - A befoglaló doboz vizsgálat már ütközést jelez!

Pixel szintű ütközésvizsgálat

- **A helyes megoldás algoritmusa:**
  - meg kell vizsgálni, hogy a két objektum területének vannak-e egymást fedő pontjai.
  - **Triviális megoldás:** a két objektum minden pixelét megvizsgáljuk
    - Számításigényes, sok ciklus
  - **Optimális megoldás:** a befoglaló doboz alapján.
    - Ha a két terület nem érintkezik, a két objektumnak nem lehet egymást fedő átlátszatlan pontja,
    - Ha egymásba lóg az objektumok doboza, meg kell vizsgálni a közös részt
    - Ehhez végig kell pásztázni annak pontjait
    - Ha találunk legalább egy olyan helyet, ahol mindkét objektum átlátszatlan, akkor ütköztek.

Pixel szintű ütközésvizsgálat

- Felhasználjuk a befoglaló dobozok által átfedett területet
- Csak ezen a területen belüli pixeleket kell átvizsgálni



Pixel szintű ütközésvizsgálat

- **Az algoritmus:**
  - Az algoritmusnak az ABW és ABH területet kell pixelenként átvizsgálni
  - Addig míg nem talál mindkét objektum képi leképzésénél legalább egy darab nem átlátszó pixelt.
- Mi okozza nagy számítási igényt?
  - A dupla ciklus, ami végighalad a sprite-ok képpontjain.
  - Minden pont értékét a központi memóriából le kell kérni, majd összehasonlítani egymással.
  - Kis méretű sprite-ok esetén ez nem okoz nagy gondot,
  - Nagyobb méret esetén jelenős erőforrást igényel
    - dupla ciklusba ágyazott feltételes utasítás végrehajtása minden megjelenítési frame-ben.

Pixel szintű ütközésvizsgálat (példa)

```
for (i=0; i < over_height; i++) {
        for (j=0; j < over_width; j++) {
            if (pixel1 > 0) && (pixel2 > 0) return true;
            pixel1++;
            pixel2++;
          }
          pixel1 += (object1->width - over_width);
          pixel2 += (object2->width - over_width);
    }
```

EGYÉB KIEGÉSZÍTŐ MEGOLDÁSOK…

Kiegészítő megoldások

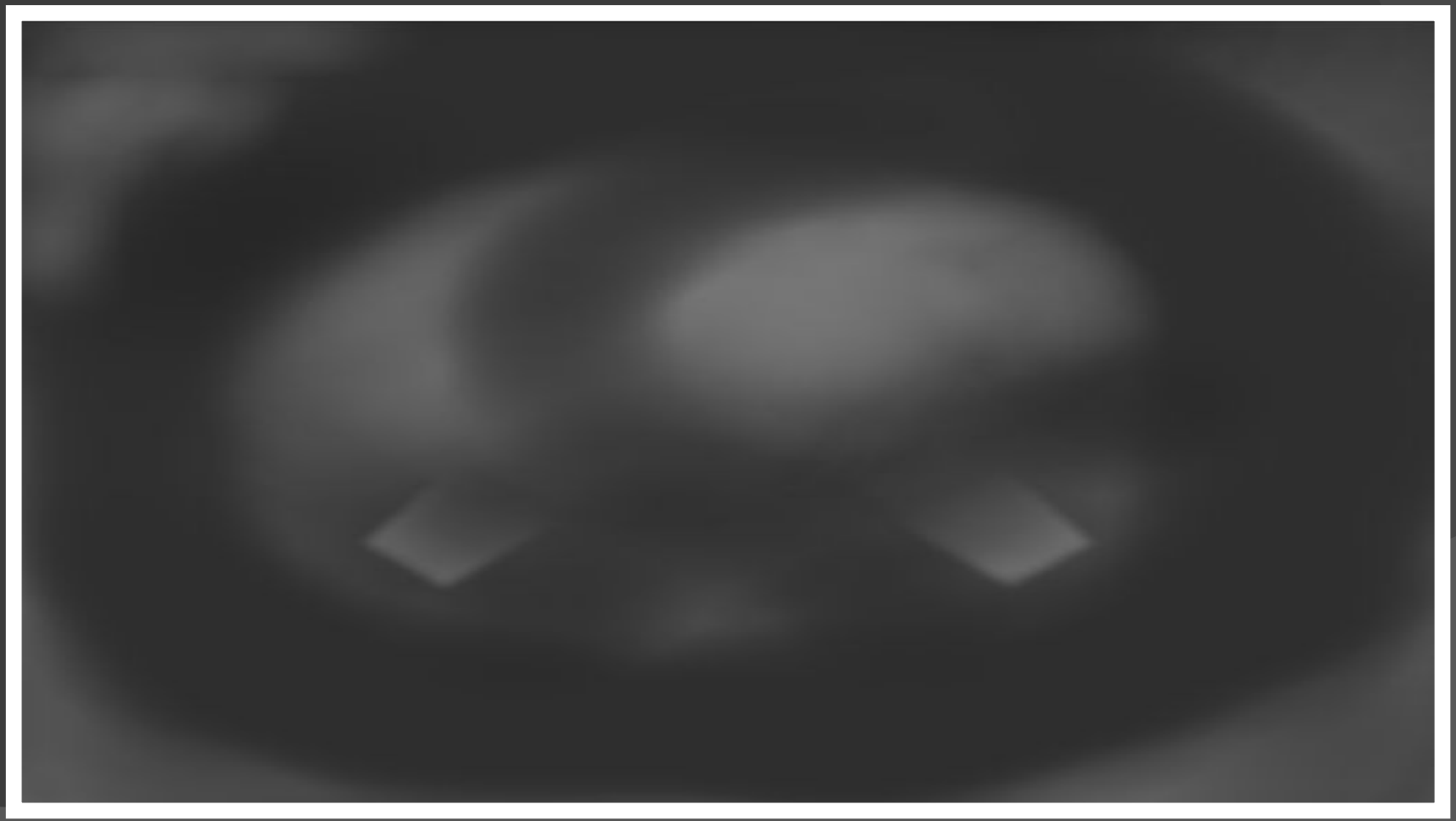
- Az irodalomban számos egyéb megoldás is kialakult
  - Általában szoftverre specifikus eljárások
- **Példa:** a **bitmaszk alapú pixeles ütközésvizsgálat**
- **Lényege:**
  - a megoldásnál egy fekete fehér képét készítik el az objektumnak
    - pl. pálya ahol mehet a motor
  - Ütközésvizsgálat esetén ezt a 0 és 1 értéket tartalmazó bitképet vizsgálják.
- **Előnye:**
  - bitek jelzik az ütközési területet,
  - így kevesebb helyet foglalnak el a memóriában, mintha RGBA kép lenne
    - RGBA esetén egy integer-ként tárolva egy pixelt tudunk feldolgozni, a bitmaszk alapú megvalósításnál 4 darabot

Bitmaszk alapú ütközésvizsgálat (példa)

- Normál játéktér

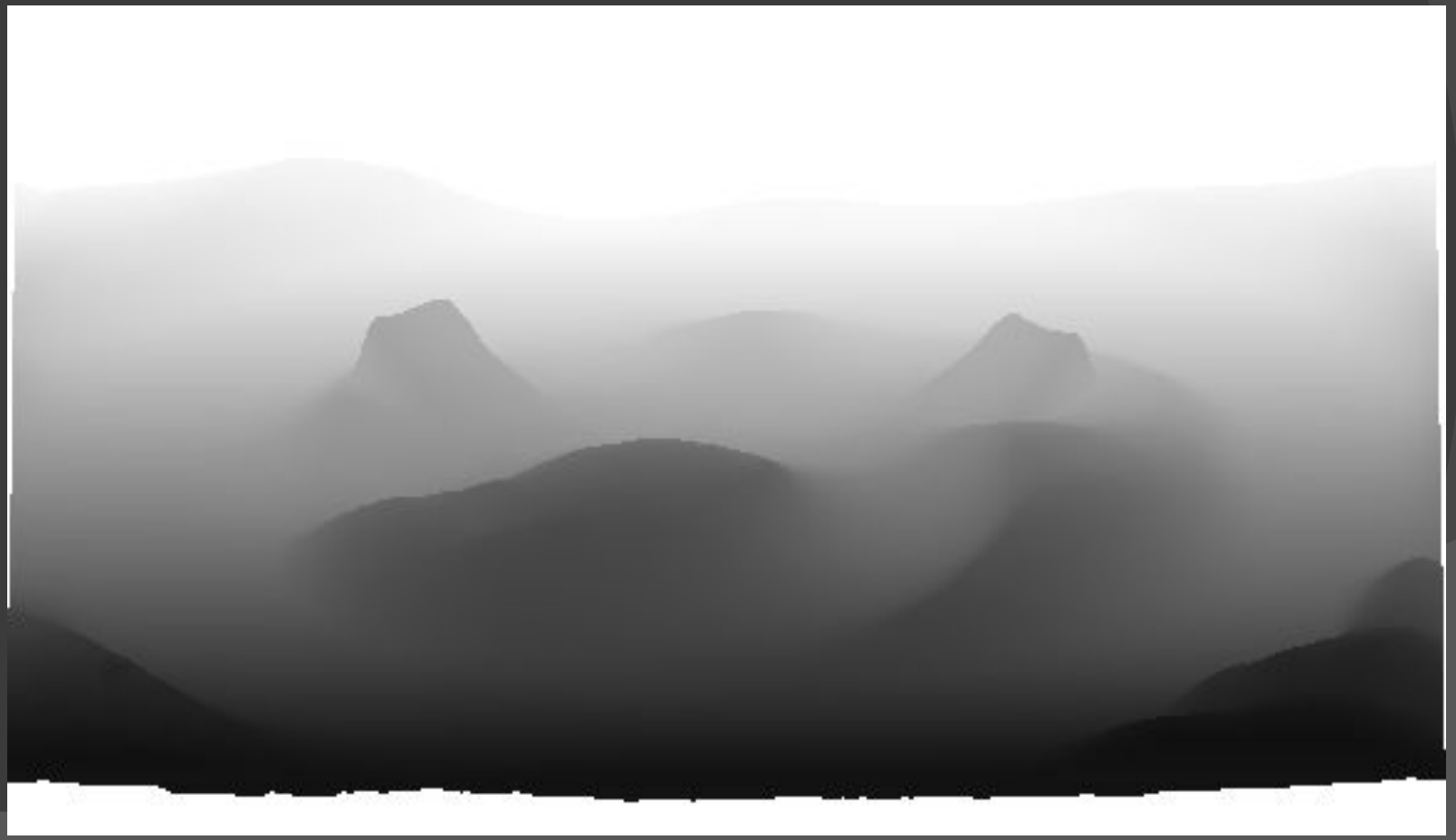
Bitmaszk alapú ütközésvizsgálat (példa)

- Objektumok mozgástere fekete fehér textúraként



Bitmaszk alapú ütközésvizsgálat (példa)

- Mélység térkép a játéktérhez



Bitmaszk alapú ütközésvizsgálat (példa)

- Valós játékmenet

GAME OVER