Production Systems and Information Engineering
Volume 11 (3), pp. 27–50
doi: 10.32968/psaie.2023.3.3

PRACTICAL GUIDE TO IMPLEMENT A SIMPLE 2D GAME ENGINE

Péter Mileff
University of Miskolc, Hungary
Department of Information Engineering
mileff@iit.uni-miskolc.hu

Abstract. Computer visualization now represents a very important area in people's lives. It has appeared in practically every area, and is now essential. Within the field, game development has grown to become a huge industry. Creating a high-quality 2D or 3D game today requires extremely serious development work. Although many platforms, developer APIs, and game engines have emerged over the years to greatly assist developers in their work, there are many people who want to understand how the technology behind games works. Therefore, they build the specific software components themselves, from which they build a complex system and write the algorithms necessary for image synthesis and other areas. This publication focuses on this area. Its purpose is to present the basic elementary components that are necessary to implement a two-dimensional game, as well as to serve as a good basis for the creation of additional higher-level structures.

Keywords: Game development, virtual world, game engine

1. Introduction

Two-dimensional visualization plays an important role in addition to today's modern three-dimensional rendering. It can be said that although the world has moved in the direction of 3D in the field of visualization in recent years, two-dimensional solutions have always been present as an additional technique and will continue to be so in the future. Several layers of computer applications belong here. Any software that has some kind of graphical interface or windowing system, and two-dimensional computer games are big representatives of the field. From the point of view of the menu system, there are always attempts to make these subsystems more visual in three-dimensional space, although they work, but mostly these solutions return to 2D rendering.

For these systems, rendering speed is not critical compared to today's systems performance. Mostly they use a small number of static images, but other transformations (rotate, scale) are not very typical. However, the opposite is true for computer games. In the case of such software, it is usually necessary to rasterize a large amount of continuously changing objects, which requires significant resources, changed dynamically depending on the moved elements. A typical feature of today's systems is the dynamic application of a large set of textures, animations and transformations in order to achieve a higher user experience. In accordance with the requirements, the use of high-quality textures with a large screen resolution is now essential, which further increases the required performance [20].

The popularity of the gaming industry has grown steadily over the last 20 years and this process is still unbroken. One of the driving forces behind this is the rapid technological evolution. Today, a mid-range mobile device/tablet can perform better than, an old Pentium 4 or a Dualcore Pentium. However, these devices are reaching many more people than before. The leap-like evolution of technology is followed by games as well. The development is clearly visible both in terms of visuals, complexity and gaming experience. Expectations are getting higher and higher, and developing hardware provides a good basis for this. Game development and modern visualization appeared on several platforms. For example, HTML5-based solutions. As a result, the number of active (underage) players has increased in recent years.

There are several opportunities for developing games:

Development with a game editor: today there are many software available that essentially fulfill the role of a game editor (Roblox, Unity, Game Editor, Construct 3, GDevelop, etc). The most important goal here is to develop software editors of a level that can be used to visually assemble the game. Of course, these editors leave the opportunity that the developers can implement unique functions that is not directly supported by the editor. The common solution for these is usually that to create a code or script written in some programming language (e.g. Javascript, `C#`, etc.) for the components and objects in question. Although the development of this kind of approach is a natural part of evolution, for many programmers the approach when we do not see the entire code base at once is strange, and it loses a little of its charm among classic game developers.

Development with a game engine: game engines have existed almost since the beginning of the PC era of computer visualization. Their task is to help the game development process to a great extent by offering ready-made, framework-level solutions for many things. Many professional game engines are now available, which integrate a lot of knowledge and work. It can be safely declared that most of them offer high quality visualization and optimized performance. That is why it is no coincidence that many engines are not available for free. The price of a really good game engine can even be very high. Game engines provide API level support. These libraries can be linked to the program to access their functions. Another characteristic - especially for three-dimensional engines - is that they have some kind of editor (2D or 3D). These interfaces do not play the role of the game editor, rather they are additional tools for editing tracks, setting material properties, or even other things.

Native development: the last category includes those developments where we do not explicitly use any external game engine, instead the developers create (almost) everything themselves in-house. Naturally, this requires the most complex knowledge and development. However, in return, the developers gain knowledge that will help them understand the details of the visualization. For this kind of development they often build their own game engine over time, which already provides a high level of experience from the point of view of software design and development.

The two-dimensional computer game world is made up of various elements. In practice, at the implementation and planning level, there can be many implementations of these, but it is a basic requirement that the elements of the world can be implemented relying on some reusable elements. This kind of implementation will make it possible for the programmers to be able to progress productively with the development of the game with the help of efficient and dynamic development.

In this paper, we examine the structure of the game world in detail. What are the important elements that are absolutely necessary and can be relied on when creating a more complicated game or even a game engine. We present a proposed design structure that can offer a functional, efficient and good basis for a more complex game engine in the future. The detailed structure is based on native development, apart from OpenGL or DirectX, there are no additional API layers that would support visualization or game creation with extra functions. Furthermore, the paper only deals with the higher-level structures required for visualization, it assumes that the lower-level building blocks (existence of a 2D texture drawing shader, matrices, image loading, input management) that are necessary for operation and visualization are available.

2. Basic elements of the virtual world

Anyone who has ever worked with computer game development knows that the first milestone for graphic applications is always being able to draw something on the screen. (Assuming that we are talking about native development). Today, in many cases, this is not even trivial, because the entry level of computer visualization has increased significantly. Today, creating an API-level "Hello World" application is not easy either, since developers have to deal with geometry, matrices, shaders and other tasks even in the case of a primitive demo, and the two-dimensional visualization is no exception.

In the following, we present the basic elements of the 2D virtual world, which can serve as a basis for the development of a two-dimensional game engine. The presented sample codes are implemented in C++ language.

2.1. Simple Texture Object

Graphics APIs work with so-called textures. For us, this means a two-dimensional (usually with alpha channel) image. The image must be loaded into main memory before use and then into GPU memory. Of course, the size of the GPU memory is limited, so we cannot usually load everything there. In most cases, this is why the games only load the graphical data for the different levels into the GPU memory when they are used (e.g. before each level). If w want to store a lot of graphical data in the GPU memory, we can use texture compression or *GPU Level streaming.*

A graphical application requires static elements in many places. Examples include a background image, a moving cloud, or even menu buttons. One of the basic elements of the game engine is therefore a static object on which more complex elements can be built later. A simple static object, if polygon-based rasterization is used, is usually made up of two triangles (Figure 1). Because it is stored in the GPU, an API specific part is needed to handle this. For OpenGL, this is the *Vertex Array Object.* In the sample code we refer this part as *CVAOobject*, which is a class for supporting those VAO related function.

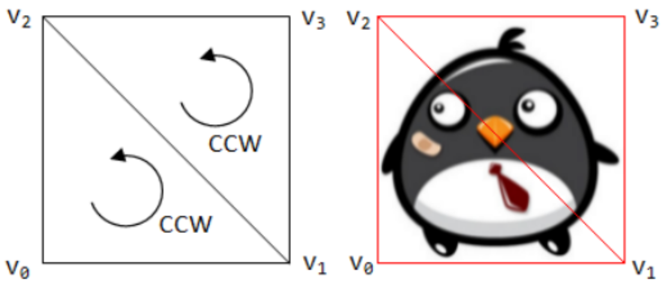

Figure 1. A common triangle-based form of handling for two-dimensional images

When entering the vertex data, two main approaches were developed, depending on what the anchor point of the image should be. Figure 2. shows the two most commonly used forms.

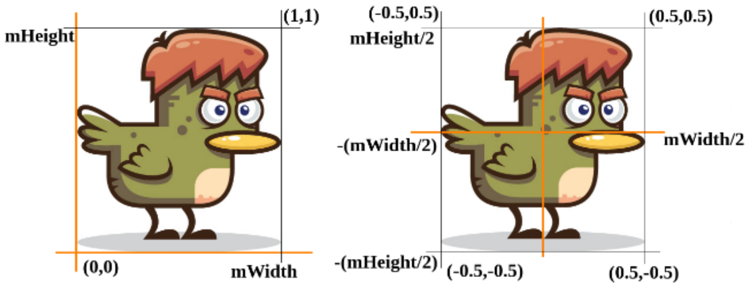

Figure 2. Popular coordinate systems

Both solutions have their advantages and disadvantages. The placement affects the drawing logic. In the first case, if we expose the object to a coordinate, the drawing will take place to the right and upwards from the point indicated by the origin of the drawing's coordinate system. In the second case, the specified position will be the center of the object. The anchor point also affects the rotation of the object, namely in such a way that the anchor point becomes the center of rotation. Of course, if we choose the solution on the left, even in the case of rotation, we can define the center of rotation and perform the calculation.

```
class CTexture {

    CVector2 mPosition;           // position of the texture
    CVector2 mRotation;           // orientation of the texture
    CVector2 mScale;              // scale of the texture
    bool bVisible;                // Texture is visible or not
    float m_WidthOriginal;        // Original width of the texture
    float m_HeightOriginal;       // Original height of the texture
    CVAOobject* mTextureVAO;      // Holds Vertex, Texture and Color
    sColor sColor;                // Color information
    string mFilename;              // Holds the filename of the texture
    string mName;                  // Material name
    float mWidth;                  // Width of the texture
    float mHeight;                 // Height of the texture
    unsigned int mTextureID;      // Holds the texture ID
    unsigned int mID;             // Global (engine) ID of the texture

public:
    CTexture();
    // Overloaded operator
    CTexture& operator= (const CTexture& _texture);
    Draw(...);
    [...]
};
```

The constructed class contains the most important basic properties required for displaying a two-dimensional image/texture. It will be just as suitable for displaying the image of a game object as it is for a menu item. In addition to the basic elements, it is advisable to have a place for a "copy function". Due to the C++ nature of the code, here we assign this role to an overloaded equals operator, which is necessary to be able to pass the data of a *CTexture* class to another class with a series of calls. A specific example can be the shooting process that appears in arcade games. When objects fire projectiles, we create a new projectile class instance (possibly choosing from a pre-created pool of projectiles). This new projectile can be filled with data very easily if we copy an existing one (the mother object). Without redefining the copy constructor/operator, the data would have to be transferred one by one to the new class, which would reduce the quality of the code.

2.2. Collision detection

An essential element of games is the object interaction: i.e. the detection when two objects collide with each other. In fact, this is not only specific to the world of games, but the same principles are also applied when, for example, we move the mouse towards a menu item. Of course, in computer games, the proper and exact level of collision detection plays a dominant role, since the game experience is formed as a result of these interactions.

In a very simplified way, the essence of collision detection is to somehow algorithmically detect whether the two-dimensional images of two or more objects overlap each other. To be more precise, the problem is a bit more complicated than that: it means whether an object has a pixel that overlaps a pixel of another object.

During the development of a game, there will surely come an important moment when we have to decide what collision detection system and model to use. The decision is not always easy and straightforward. There are some types of games where the interactions can be very complex, and often not all problems can be seen in advance. Nevertheless, the applied model is important, as it has a great impact on the development time and the game experience itself. Basically, collision detection systems can be classified into the following two groups:

- **Pixel-based collision detection:** detects the overlap of the pixels of the images belonging to the collided objects. It can detect a precise, real collision.

- **Enclosing shape-based collision detection:** The overlapping of objects is not determined at the pixel level, but at the level of some enclosing object(s) (box, circle, polygon, etc.). It usually doesn't allow precise collision.

Pixel-based collision detection can be computationally intensive and complicated, depending on the complexity of the texture associated with the object. Moreover, the texture of the object is stored in the GPU, so a proper pixel level collision implementation is not simple. For this reason, where possible, game developers try to enclose the moved elements into some object(s) and perform the collision analysis on this. The most common objects are the circle and the box, because they are very simple elements. The subsequent calculations with them (collision test, rotation, translate, scale, etc.) are not nearly as computationally intensive as, for example, in the case of an enclosing polygon or the pixel-level test. Although they do not approximate the object well, they are still effective and can be used well in practice.

2.2.1. Bounding box based collision

One of the simplest, yet most popular forms of collision detection is the bounding box-based solution (rectangular collision detection). In this case, the object is surrounded by a "box", i.e. a square or rectangle. If the shape aligned to the base axes of the scene, it is called the Axis-aligned Bounding Box (ABB).

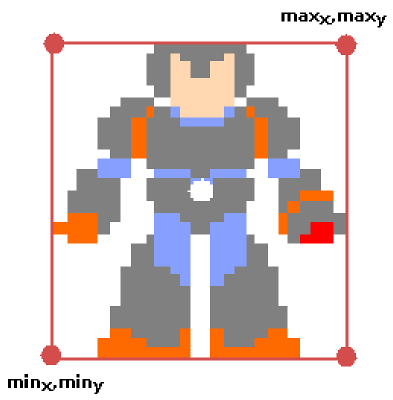

Figure 3. Best-fit containment box. There are apparently no empty pixels between the outermost points of the shape and the sides of the box

In the simplest case, the bounding box is determined by the object's two-dimensional image and texture. This can be calculated very simply during their load process: the width and height of the texture image will determine the box. When defining the box, they usually try to determine or specify the best fit rectangle. The reason for this is to reduce/avoid false collisions. That is why so important to draw the Sprite correctly and to avoid unnecessary transparent pixels on the edges. Just think about the fact that if we increased the size of the box along the x axis in the case of the above object, it would result for us sensing a collision even when we have not yet reached the wall.

The following code snippet shows a possible bounding box implementation:

```
class CboundingBox2D {

  CVector2 minpoint;              // Box minpoint
  CVector2 maxpoint;              // Box maxpoint
  CVector2 bbPoints[4];           // bounding box points
  float boxHalfWidth;             // box half width
  float boxHalfHeight;            // box half height
  matrix4x4f tMatrix;             // Transformation matrix
  bool mEnabled;                  // BB is enabled or not

public:
...
};
```

In two dimensions, a bounding box can be specified with the four corner points (*bbPoints[4]*), but in order to speed up later calculations, it is advisable to store the minimum (*minpoint*) and maximum point (*maxpoint*) relative to the screen coordinate system. In the figure above, this means the lower left and upper right points. On top of all that, the calculations will require a matrix class that performs the transformation, and for performance considerations, it is worth storing the half width and half height of the box. Because we use the left-bottom point of the image as the origo of the texture coordinate space, these values are necessary during the center based rotation. When moving the object (translate, rotate, scale), the coordinates of the box need also to be transformed. We need to do this when we calculate the new position of the object when it moves. Another solution could be to calculate the points of the box when the program will use it (e.g. collision detection, visibility test, etc.), but in this case more resources are needed for multiple calculations.

The collision determination algorithm is very simple to formulate: when two bounding boxes of the objects overlap each other, the objects collide. Figure 4. illustrates this.

It is clearly visible that the fact of the collision can be clearly determined from the overlap of the enclosing boxes. From the point of view of implementation, to save the CPU from unnecessary calculations, a more common solution is to detect when there are no collisions:

```
boolean CheckBoxOverLap(CBoundingBox2D box1, CboundingBox2D box2) {
```

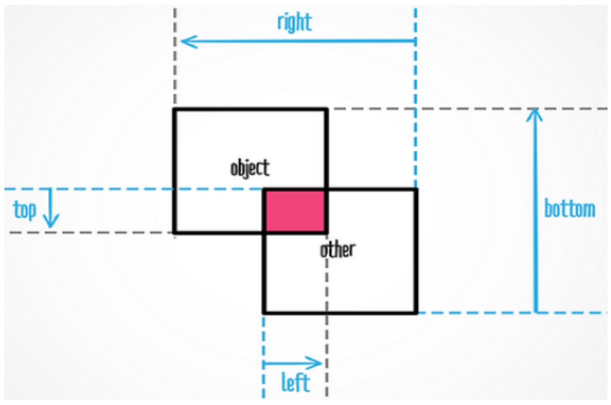

Figure 4. AABB collision

```
        if (box1.maxpoint.x < box2.minpoint.x ||
                box1.minpoint.x > box2.maxpoint.x){
            return false;
        }

        if (box1.maxpoint.y < box2.minpoint.y ||
           box1.minPoint.y > box2.maxpoint.y){
           return false;
        }
        return true;
}
```

This paper does not cover the rotation of the bounding box. While the OBB (Oriented Bounding Box) type solution gives really good results [7], in practice the AABB-based rotation is also satisfactory in most cases. In the case of AABBs, the box (its corners) must also be rotated and calculated during rotation. And from the new four corner points, an AABB can again be built around the object. The downside is that the size of the box changes, therefore it affects the accuracy of the collision test.

2.3. Sprite animation

Animation plays an important role in computer visualization. This makes the application really "live", be it the animation of a menu, window or a jumping figure. In the case of computer games, a lot of emphasis is placed on the development of continuous, high-quality movements. The classic, well-known solution to creating an animation is essentially nothing more than alternating the set of images (phases) that make up the animation in a specified sequence and speed. Images can be called an array of textures loaded into memory, which contains each phase of the animation. Many people call this texture object a Sprite. The more phases the array contains, the more continuous the object's animation will be when displayed. In the file system, the images of the animation can be stored in several ways. The most common solution is to store individual frames next to each other in a larger image (*spritesheet*). Figure 5. illustrates this:

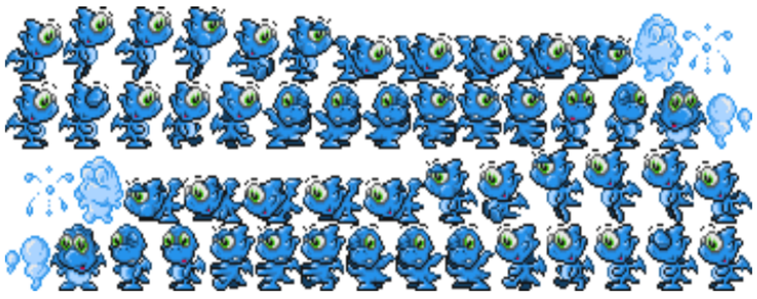

Figure 5. Sample spritesheet

Creators choose a uniform background color and store the animations next to each other. When loaded, these are split into separate texture objects. Two-dimensional images of this kind are collectively called "*Pixel Art*" (Pixel graphics), because they are mostly made by hand, drawn pixel by pixel. Although modern three-dimensional graphics have greatly transformed computer visualization, many pixel graphics games are still being made today.

```
class CSprite {

  vector<CSpriteFrame*> mFrames;  // Frames vector
  CVector2 mPosition;              // Position of the sprite
  CVector2 mScale;                // Scale of the sprite
  int mNumFrames;                 // Number of frames
  int mActualFrame;               // Actual frame
  unsigned int mLastUpdate;       // The last time the animation was update
  unsigned int mFps;              // The number of frames per second
  float mRotationZ;               // Z oriented Orientation of the sprite
  string mName;                   // Name of the Sprite
  bool mLoopAnimation;            // Loop animation

public:

...
};
```

As you can see from the code, the *CSprite* class is nothing but a unit that stores the frames and the properties belonging to the class. Each animation phase is represented by a *CSpriteFrame*, which will be described in more detail later.

The additional data stores important properties such as the sprite's position (*mPosition*), size (*mScale*), name (*mName*), number of frames, as well as other data that will be needed to actually move the phases one after the other. The name is important because it is much easier to refer to an animation phase by name in a program, it makes it user-friendly. For example, request the "running" phase from the engine.

It is clear that the animation phases are not stored directly in a *CTexture2D*-based vector, but are implemented through a *CSpriteFrame* class. The question may arise, why is this actually necessary? The answer has to do with bounding boxes. The phases of the game objects can be of different sizes (e.g. the main character falls to the ground, jump, shoot). It goes without saying that the collision detection will have to be done accordingly. Since the *CTexture2D* class is a basic unit, it is not advisable to endow it with some enclosing object that helps collision detection. This task is performed by the *CSpriteFrame* class, whose simple structure is as follows:

```
class CSpriteFrame {

  CTexture2D* mFrame;                   // Frame texture
  string mName;                         // Name of the frame
  CBoundingBox2D* mBBoxOriginal;        // Original Bounding box
  CBoundingBox2D* mBBoxTransformed;     // Transformed Bounding box

public:
  CSpriteFrame();

  CSpriteFrame& operator= (const CSpriteFrame& _spriteFrame);
...
}
```

The *CSpriteFrame* class is therefore responsible for storing different frames. The image is stored in a class of type *CTexture2D*. It is also advisable to store the name of the frame, it may be needed in certain situations. The last important piece of data is the frame's bounding box, which will be needed during the collision test. We store two versions of the box: one is the original and the other is the transformed one. The importance of storing both is to be found in performance. The game engine needs the box of the object multiple times. Since the object can rotate, the box will also transform with it. For this reason, it is advisable to store the current transformed result as well.

Although until now only animations of game objects have been in the foreground, the *Sprite* class itself is just as suitable for describing moving GUI elements. A good example of this is a button that changes when the mouse hovers over it. A *Button* class can essentially be an element based on a Sprite object with two (or more) animation phases.

2.3.1. Animation description file

The sprite loading process should be designed and supported based on a description file. This approach supports the previously emphasized flexible programming API design concept. As a basis for the format, it is advisable to choose some well-known storage format such as JSON or XML. An effective sample format can be the following XML form:

```
<?xml version="1.0" encoding="utf-8"?>
<animation name="Yoshi_anim">
        <sprite numofframes="4">
                <frame name="Yoshi_Anim_Start" file="yoshi1.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
                <frame name="Yoshi_Anim_Start" file="yoshi2.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
                <frame name="Yoshi_Anim_Start" file="yoshi3.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
                <frame name="Yoshi_Anim_Start" file="yoshi4.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
        </sprite>
</animation>
```

The example shows an XML based description format. The format is apparently simple: we can define any number of frames, to which we can associate an enclosing box and a name. However, do not forget that the descriptor alone will not be sufficient to store the animations of a more complex object, since it can only store one phase of the object.

2.4. The Game Object

The Sprite class alone is not enough for everything. In order to be able to comfortably describe the game elements and objects with high-level elements during the programming of a game, it is necessary to introduce a "*game object*" class. The *Sprite* class is still needed, as we can use it for example to create GUI elements (e.g. animated buttons, etc.) or whatever we want, as a basis for the actual game objects. In a two-dimensional computer game, a game object has not only one animation phase, but as many states as it can take. For example, the main character can run, throw, jump, etc. If we think about it, we need a higher level class that contains an array of *Sprites*, and depending on the state of the object (walking, crouching, etc.) they can be changed. In addition to all this, it is of course also necessary to introduce additional state variables and methods. An example game object implementation can be:

```
class CGameObject2D {
        string mName;                     // Entity Name
        CVector2 mPosition;               // Position of the object
        CVector2 mDirection;              // Direction of the movement
        CVector2 mScale;                  // Size value
        vector<CSprite*> mAnimations;     // Animation
        float mSpeed;                     // Speed of the object
        float mRotation;                  // Rotation value
        bool mVisible;                    // Visible or not
        bool mCollidable;                 // Object is collidable or not
        bool mInFrustum;                  // Object is in screen frustum or not
        unsigned int mCurrentAnim;        // Current Animation Frame
        unsigned int mNumberOfFrames;     // Number of Animations
        unsigned int mID;                 // ID of the Object
        int mZindex;                      // z index of the object
        C2DGraphicsLayer* mParentLayer;   // Parent layer of the object

public:
...
};
```

The first and most important thing that a GameObject class needs is the name (*mName*) and identifier (*mID*), because we will be able to refer to it with these. The mID alone would be enough, but from a convenience point of view, it is better if we can also refer to the object by name. Of course, an object needs position, direction, and properties that determine its dimensions. The direction can be used to implement a kind of automatic movement. So the user provides the direction and speed (*mSpeed*) and the game engine does the job of getting the object to the target. The animation phases belonging to the object are stored in the array of sprites (*mAnimations*), and the rotation of the object is stored in mRotation. Although the optimization of the visualization is not part of this article, from the point of view of visibility, two useful data are also displayed here. With *mVisible*, the programmer can control whether the given object should be drawn or not. And the *mInFrustum* data member is an internal variable reserved for the purpose of storing the information that the engine determines if the object is within the screen range. It needs to be calculated after updating the position of the object and before the drawing occurs.

A particularly useful data member is the *mZindex*, which can be used to determine the order in which objects are drawn. In certain situations, the objects can overlap each other, which usually means an order determined by some programming logic. There are objects (e.g. cloud) that are drawn on top of, for example, the location object. The implementation of this requires the introduction of a numerical value (e.g. z value) which is meant to represent this sequence. An implementation logic could be the following: the smaller the z value of the object, the closer it is to the viewer, i.e. the later it is drawn. However, the implementation requires an additional addition, according to which the objects must be sorted based on the z value before drawing. The correct order of drawing can be ensured in this way.

3. The layering system

If we observe several popular 2D games, we can discover that certain parts of the screen, the background, and possibly some non-game objects move at different speeds. In practice, we can say that the elements are organized into so-called logical layers. It cannot be said that all games work this way, but the fact is that the introduction of logical layers greatly helps the construction of game worlds, moving objects, etc. The most common division in the case of a simpler game is to separate a layer for the background of the level (e.g. super mario), a layer for the real game objects (e.g. mario, turtles, etc.) and possibly a layer for the things that are moved in the foreground. Of course, any number of layers can be specified, thus increasing the visual appearance of the game. In the image below, we can see three logical layer-based implementations:

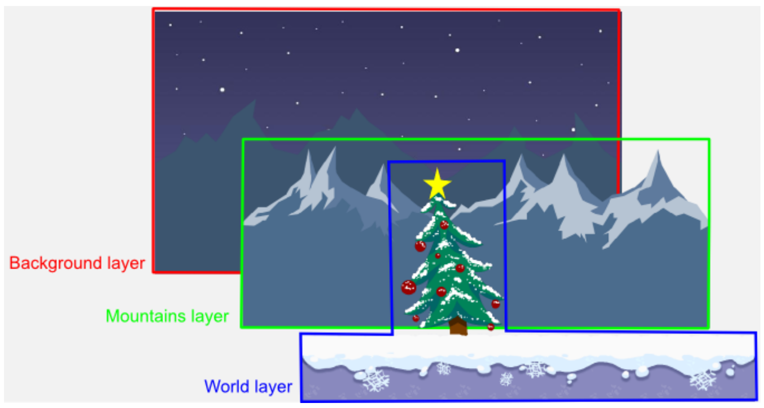

Figure 6. A game scene that implements three layers

Based on these, a class capable of storing and managing layers can be designed. The implementation of the class can be as follows:

```
class C2DGraphicsLayer {

  vector<CGameObject2D*>  mObjectList;
  vector<CTexture2D*>     mTextures;
  bool mVisible;                        // Is layer visible or not
  bool mEnableCollision;                // Enable Collision on Layer
  CCamera2D *mCamera;                   // Camera for the layer
  string mName;                         // Layer name
  int mID;                              // Layer ID
  C2DScene* mParentScene;               // Parent Scene
public:
...
};
```

Its interpretation is very simple: a list (*mObjectList*) is needed for storing the game objects. In addition to all of this, we also consider it advisable to introduce a list (*mTextures*) that only stores 2D textures. These are simple images such as game backgrounds, clouds, etc. that are not animated. Although the game object itself would be able to store them as a so-called single-phase Sprite, storing the simple textures themselves is simpler and more efficient. Since the purpose of the layer is to be able to store virtually any element that can appear in the 2D world, the class can be optionally extended later with elements such as Tilemaps, Particles, etc. The class contains the reference to the scene (*mParentScene*) to which the layer itself will belong (see later). In addition, the property that switches visibility on and off (*mVisible*), the name of the layer (*mName*) and its identifier (*mID*) are absolutely necessary.

With the above layering technique, the so-called *parallax scrolling* effect, which is popularly used in two-dimensional games in order to increase the visual experience, can be realized. The basic idea is to divide the game world into layers and then move the layers at different speeds. Moving the layers does not mean to change the coordinate of every layer object. A common solution for this movement is to use a camera (*CCamera2D*) implementation, which uses the view matrix to achieve the movement. Similar to any 2D platformer and strategy games where the map can be traversed (Warcraft, etc.) by moving the mouse.

Another problem appears with the introduction of the camera. Since the actual coordinate of the object (*mPosition*) does not change, it is only moved to a different location by the view matrix when drawing, so a method that can calculate the virtual spatial location of the object for CPU tasks such as collision detection must be introduced. For example, in the case of platformer games, the main character always stays in the range of the screen, while he has already gone several screens away in the world.

4. The level scene

The introduction of layers helps a lot to really be able to handle a more complex game world. However, this is not the end, as it is advisable to include the layers in some structure. A possible implementation of these is the game scene (*Scene*). If we want to define it, then the scene includes a detail taken out of the current game world. This can be a complete track, or even a part of it. If we remember the classic super mario game, we could see that we could slide down underground on certain pipes. The easiest way to achieve this is to create the current level from several scenes: one for the regular level and a separate scene for each underground part.

The implementation of the scene is very simple, because it actually acts as an enclosing logical structure. The following code shows such a sample:

```
class C2DScene {

  CVector<C2DGraphicsLayer*> mLayers;     // Layer for objects
  string mName;                           // Name of the scene
  bool mVisible;                          // Visibility flag
  CCamera2D *mSceneCamera;                // Global camera for the whole scene

public:
...
}
```

The scene class includes different layers (*mLayers*) so that different parts of the game can also enjoy the benefits of layered implementation. We need to store the name of the scene (*mName*), which, like before, has practical reasons. To turn the scene on, a data member must be introduced (*mVisible*). Don't forget that because super mario was jumping in the upper part of the track, actually the underground part was also in the memory, just not visible. Finally, it is advisable to introduce a scene-level camera (*mSceneCamera*). Although each layer has its own camera, in many cases the need arises to move the entire scene together.

The class should not be strong in data members, but rather in the provided services and methods. We need to be able to add, delete and query layers. The class should provide high-level services such as getting the reference of the game objects on whatever layer they are, freeing them if, for example, we need to delete a projectile, handling camera movements, z-based sorting, etc. These do not mean a complicated implementation, rather they just put the proper interface to the hands of the programmer. The use of this interface, the class itself, will be effective if it provides convenient access to the layers included in it.

4.1. Scene manager

The realization of the scene becomes complete when we can manage them. And for this, an additional enclosing class is needed. Its role is practically storing the scenes (*mScenes*) and providing a number of convenience functions and interfaces with which we can access the stored scenes or game objects either by *id* or by *name*. An example implementation is:

```
class C2DSceneManager {
  vector<C2DScene*> mScenes;
  bool mDrawBoundingBox;           // Draw BB or not
  sColor mBoundingShapeColor;      // Color of the bounding shape

public:
  ...
     C2DScene* LoadSceneXML(string sceneFilename);
     void RegisterScene(C2DScene* scene);
     void Render();
     CGameObject2D* GetObjectByName(string name);
     C2DScene* GetScene(u32 sceneID);
     C2DScene* GetSceneByName(string name);
     void FreeObject(u32 id);
     void FreeObjects();
     void Clear();
     void FreeAScene(string name);
     void FreeASceneByID(u32 sceneID);
     ...
}
```

The implementation of the class will not be complicated, its role is more limited to the service functions. The most important of these have been shown in this sample.

4.2. Storing of the virtual world

Even in the case of the simplest computer game, there is a need for the virtual world not to be burned in and stored in the various classes, but for some solution to be created to manage it effectively. It is obvious to store the different "levels" in the file system. The two common ways to achieve this are the binary and text storage forms. For beginners, the text format is absolutely recommended, since when there is no mature game idea or code to drive the game development, the text file-based storage greatly facilitates continuous modifications and experiments, as it can be edited with a simple text editor.

Below is an XML description of a sample virtual world:

```
<?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="sky.pcx" />
        </layer>
        <layer id="1" name="Mountain_Layer">
                <texture id="1" x="0" y="0" file="mountain.tga" />
        </layer>
        <layer id="2" name="Ground_Layer">
                <texture id="2" x="0" y="0" file="ground.tga" />
        </layer>
        <layer id="3" name="Character_layer">
                <gameobject id="777" name="Liza" collidable="1" zindex="0">
                        <sprite file="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>
```

The sample describes a simple world with three layers and one movable character. The girl.ani in the description describes an animated character whose format is the XML animation descriptor outlined above. Of course, the world description above contains only the most important elements, and can be easily extended with new features as needed.

5. The heart of the engine

The logical structure of a software must always be designed and built based on software technology aspects. The structural arrangement, the relations of the classes (provided they are written in OO language) cannot be approached exactly. Since there are many design patterns and subjective factors can greatly influence the structure, there is no uniform direction or system of rules that need to be followed in any way when designing the structure of our software. Of course, it is always worth looking around, as there are well-established patterns that may differ depending on the language and the frameworks used.

Properly designing the basics of the software can require serious knowledge in this direction, especially when it comes to your own game engine. Creating the right environment helps a lot in making the development of game classes to be easy. A typical indicator of an inadequate design is when the developer needs to spend a lot of time in the program code to find where the given information comes, what calls the method and where certain elements of the logic are located.

The following simple C++ code shows an example of the structural basics, which is enough to start creating a program on it. Game softwares today is typically developed in an object-oriented manner, so they follow similar patterns in structure. Somewhere deep in the software, a pattern like the one below is often applied:

```
/// Universal Application Class
class App {

protected:
CEngine *mEngine;           // Engine Class as member variable

public:
App();
virtual ~App();
virtual void Startup() = 0;
virtual void Run() = 0;
virtual void Shutdown() = 0;
virtual void ResizeWindow(int width, int height) = 0;
};
```

The above class provides a general framework for our future application. Since the methods are apparently pure virtual, they will be implemented by the child class. And the instantiation placed in the constructor enables the automatic initialization of the (game) engine, and the destructor performs its release:

```
CApp::CApp(){
    mEngine = nullptr;

    // Allocate Memory for the ENGINE
    mEngine = new CEngine();

    if (mEngine == nullptr) {
        printf("\nError: cannot allocate memory for Engine!");
    }
}

CApp::~CApp() {
    if (mEngine != nullptr)
        delete mEngine;
}
```

It is obvious that because the methods are pure virtual, they will be implemented by the child class. So, in order to use the above class, a specific class (a game class) should be created by inheriting it from the *CApp* class. With the inheritance, our main game class will get the *mEngine* reference directly, which is an effective way to handle engine functions. An example game class:

MyGame.h

```
    class MyGame : public CApp {

    public:
            MyGame();
            ~MyGame();

            void Startup(void);
            void Run(void);
            void Shutdown(void);
            void ResizeWindow(int width, int height);
    };
```

In the example, the child class "*MyGame*" implements the corresponding methods of the App parent class. "*Startup*" is intended to handle the initialization parts required after the direct start of the application. For example, opening the window, creating a graphic context, and everything else that is intended here. The "*Run*" method will be responsible for the implementation of the "*game loop*", for continuous running, and the "*Shutdown*" is reserved for performing follow-up work before the application stops. For example, freeing memory areas, eliminating graphics context, freeing resources, etc. Another inherited method is "*ResizeWindow*", whose task would be to respond to the resize event of the application window.

Finally, the code that includes the missing main function:

main.cpp

```
    #include "App.h"

    int main(int argc, char *argv[]) {

        MyGame appliation; // My application

        appliation.Startup();
        appliation.Run();
        appliation.Shutdown();

    return 0;
}
```

Although the sample presented in this form is already suitable for the development of applications, in the case of more complex software and graphic engines, it is of course necessary to expand it in several directions. Therefore, in the following, we will review the extension of the above code with subsystems, for example, we can find out why it is important to assign the "virtual" qualifier to the mentioned methods of the parent class.

5.1. Subsystems and components

A computer game or game engine can consist of many different subsystems and components that communicate with each other. When the engine starts, each subsystem must be initialized, usually in a predetermined order. Certain components can also depend on each other. For example, if subsystem B depends on subsystem A, subsystem A must be started and configured first. And in the event of a shutdown, the reverse of this often has to happen. Correct implementation of such complex software requires the use of design patterns. In practice, several patterns (Command, Flyweight, Observer, Prototype, Singleton, State) are used as a basis for creating the structure. A game or engine usually uses several patterns depending on the needs to be served. In the following, we will show through a short example how the above example can be supplemented into a more efficient, somewhat engine-like, more complex structure.

It is a general expectation that the logical parts of the software should be designed in such a way that they appear in the form of independent components, minimizing the dependence on other parts. During planning, it is advisable to first proceed from the higher level units to the smaller units. So, first of all, the subsystems must be clarified, and then a comprehensive operating framework in which the subsystems are embedded is necessary for their use.

First, let's examine the principles by which a subsystem can be built.

```
class CComponent {

protected:

        int mID;          // Unique id of the component
        string mName;     // Name of the component

        public:

        CComponent();
        virtual ~CComponent();
        int getID(void);
        void setID(int id);
        void setName(string name);
        string getName(void);
        virtual void Startup(void) = 0;
        virtual void Update(void) = 0;
        virtual void Shutdown(void) = 0;
        virtual void HandleMessage(void) = 0;
};
}
```

There are common points in the structure of the subsystems, so it is advisable to highlight them in a separate class. In the present example, *CComponent* provides us those (currently minimal) elemental functions that all subsystems should know based on our first approach. Each class that will derive from it will have an ID (*mID*) and a name (*mName*) and their getter and setter methods. In addition to the data members, initialization (*Startup*), data updating (*Update*), shutdown (*Shutdown*) and message handling (*HandleMessage*) methods are also needed, which are apparently pure virtual and without a body part, so these will have to be implemented in the child class. This will make it possible to manage (start, stop) all compensations uniformly in the future.

After the proper design of the parent class, the design of the subsystems follows. The following sample shows an example for defining subsystems:

```
#include "CComponent.h"

class CSampleSubsystem : public CComponent {

public:
        CSampleSubsystem();
        ~CSampleSubsystem();
        // Component inherited tasks
        void Startup(void);
        void Update(void);
        void Shutdown(void);
        void HandleMessage(void);
};
```

The structure of the subsystem is similarly simple at the current level. During the implementation, it is mandatory to implement the inherited virtual methods, which will be needed in the later control class that unites the subsystems, as well as those methods that will already be subsystem specific (e.g. graphic context initialization, sound system configuration, etc.). There is no specific part in the current sample.

The next step is to design at least one central element or class, which will serve as the central element of the entire software structure. It unites the subsystems and provides additional important functions such as the initialization and shutdown of the subsystems and providing the so-called "main loop" (game loop). In our approach, this will be the *CEngine* class.

```
class CEngine
{
    // List of subsystems
    vector<CComponent*> mComponents;

public:

    static GraphicsManager*    gGraphicsManager;   // Graphics Manager Subsystem
    static CInputManager*      gInputManager;      // Input Manager subsystem
    static CShaderManager*     gShaderManager;     // Shader Manager subsystem
    static CTexture2DManager*  gTextureManager;    // Texture Manager
    static C2DSceneManager*    g2DSceneManager;    // 2D Scene Manager
    [...]

    CEngine();
    ~CEngine();

    void Shutdown();                                  // Shutdown
    void MainLoop();                                  // Main loop of the engine
    void RegisterSubSystem(CComponent *system);

private:

    bool InitSubSystems();            // Init all built in subsystem
};
```

The *CEngine* class is a very important part of the program. Its task is to bring together and control the available components. Here is the so-called *Main Loop / Game loop*, which is an infinite loop interpreted under conditions. This is where basic functions such as input handling, updating states, measuring elapsed time, drawing and moving objects, updating the screen, etc. are performed. In practice, due to the direct reference in a real application, it is advisable to treat the subsystems as separate, static units. Their static nature makes referencing them anywhere in the code very easy. For example, to access the graphics manager: *CEngine::gGraphicsManager*. This approach practically corresponds to the well-known "*Singleton*" design pattern, according to which only one instance of the given class can exist.

6. Conclusion

Computer game development is a process that requires complex knowledge, which also requires expertise in software design, algorithmization and graphics. Game engines try to collect and organize this huge knowledge, thus putting an effective tool in the hands of the developers. However, if we want to understand the real operation in the background, there is no other way than to build the basic components by yourself, with which certain games can already be made. On this path, we can gain a degree of programming and other experience in building more complex software systems, which is not possible in any other way. It would be advisable for all game developers to start by going down this path, and only later turn to the world of game engines.

References

[1] Akenine-möller, T., Haines, E.: *Real-Time Rendering.* A. K. Peters. 3nd Edition, 2008.

[2] MARÍN-LORA, C.; CHOVER, M.; REBOLLO, C., REMOLAR, I.: *A game development environment to make 2D games*, Communication Papers, Vol.9 – No18, pp. 7-23, 2020.

[3] Charles Kelly.: *Programming 2D Games*, A K Peters/CRC Press; 1st edition, 2012.

[4] Jason Gregory: *Game Engine Architecture*, A K Peters/CRC Press; 3rd edition, 2018.

[5] Eric L.: *Foundations of Game Engine Development*, Terathon Software LLC, 2019.

[6] Nicolas A. B.: *Hands-On Unity 2022 Game Development: Learn to use the latest Unity 2022 features to create your first video game in the simplest way possible*,Packt Publishing; 3rd edition., 2022.

[7] Steven J. V.: *2D Collision Detection Focus on OOBB/Point Collisions*, Independently published, 2019.

[8] *Gaffer On Games*, `https://gafferongames.com/post/fix_your_timestep/`, 2023.

[9] David G.: *Core HTML5 2D Game Programming*, Pearson, 2014.

[10] Alan Thorn: *Game Engine Design and Implementation*, Dave Pallai, 2010.

[11] Anis Zarrad: *Game Engine Solutions*, Simulation and Gaming, InTech, pp 75-85., 2018

[12] *Game Maker*, `https://gamemaker.io/`, 2023.

[13] *Unity Engine*, https://unity.com, 2023.

[14] *Roblox Game Platform*, https://www.roblox.com, 2023.

[15] *Constructs 3*, https://www.construct.net, 2023.

[16] *GDevelop*, https://gdevelop.io, 2023.

[17] *Phaser HTML5 game engine*, https://phaser.io, 2023.

[18] *Cocos game engine*, https://www.cocos.com, 2023.

[19] *Godot engine*, https://godotengine.org, 2023.

[20] P. Mileff, J. Dudra: *The Past and the Future of Computer Visualization*, Production Systems and Information Engineering, Volume 10, No 1, pp. 16-29., 2022. http://doi.org/10.32968/psaie.2022.1.2.

[21] Eleftheria Christopoulou, Stelios Xinogalos: *Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices*, International Journal of Serious Games, pp 21-36,m 2017. http://dx.doi.org/10.17083/ijsg.v4i4.194