Tutorial: Create 2D Game Engine using C++

URL: <https://www.youtube.com/playlist?list=PL-K0viiuJ2RctP5nlJlqmHGeh66-GOZR_>

# Purpose

This is the second half of the notes on the video tutorial. The first Word document was quite large because of the animated gifs I created using the tool ***ScreenToGif***. An animated gif shows up quite well in a Microsoft Word document and it is quite useful in showing off the progress of the work in building the platform game in the tutorial.

# 19. Collision Handling – Part I

At this point in the video series, we have a camera that follows the player around as he moves on the screen. In addition, we also modified the TextureManager::Draw() method to move the background image a bit slower than the game map – this is a common way to introduce depth in a 2D platformer. The technique is called parallax scrolling which we will cover in video #28.

In this and the next video the presenter will introduce collision. We removed the effects of gravity in a previous video and need to add it back in, but we want our Warrior to stop moving down when he hits a block.



Figure - Our Warrior missing the effects of gravity.

* Create a new class named CollisionHandler
  + Place in a new folder named Collision
  + Add \*.h and \*.cpp
  + Make it into a singleton

CollisionHander.h:

1. #ifndef COLLISIONHANDLER\_H

2. #define COLLISIONHANDLER\_H

3.

4.

5. class CollisionHandler

6. {

7. public:

8. inline static CollisionHandler\* GetInstance() {

9. return s\_Instance = (s\_Instance != nullptr) ? s\_Instance : new CollisionHandler();

10. }

11.

12.

13.

14. private:

15. CollisionHandler();

16. static CollisionHandler\* s\_Instance;

17. };

18.

19. #endif // COLLISIONHANDLER\_H

20.

CollisionHandler.cpp:

1. #include "CollisionHandler.h"

2.

3.

4. CollisionHandler\* CollisionHandler::s\_Instance = nullptr;

5.

In the game map that we created as part of video #14 we created two layers – T1 and B2. We will consider the tiles on layer T1 as the ones that will require that we detect collision.

This is our current map:

A video game with a pixelated background

AI-generated content may be incorrect.

Figure - Current game map

We will block out the B2 layer:

A blue and white line

AI-generated content may be incorrect.

Figure - Blocking out the B2 layer from our game map

The only tiles remaining to view are the ones from T1:

A screen shot of a graph

AI-generated content may be incorrect.

Figure - Our game map displaying only T1 layer tiles

When we detect a non-zero TileID on this layer – we will regard that as a solid block that the player will “collide” with in our game map. The assumption we will make in building our code is that there will be one layer (could be more) that will contain the tiles that are “solid” and will require that we check for collision with our player.

The CollisionHandler class will hold the TileLayer and TileMap of the “collision layer”. In addition, we will define two key methods:

* CheckCollision – detects if two objects have collided
* MapCollision – detects if an object/player collided with a map element

In addition, we add information about the layer we are using for collision detection.

* Update CollisionHandler.h

1. #ifndef COLLISIONHANDLER\_H

2. #define COLLISIONHANDLER\_H

3.

4. #include "SDL.h"

5. #include "TileLayer.h"

6.

7. class CollisionHandler

8. {

9. public:

10. inline static CollisionHandler\* GetInstance() {

11. return s\_Instance = (s\_Instance != nullptr) ? s\_Instance : new CollisionHandler();

12. }

13.

14. bool CheckCollision(SDL\_Rect a, SDL\_Rect b);

15. bool MapCollision(SDL\_Rect a);

16.

17. private:

18. CollisionHandler();

19. TileMap m\_CollisionTilemap;

20. TileLayer\* m\_CollisionLayer;

21.

22. static CollisionHandler\* s\_Instance;

23. };

24.

25. #endif // COLLISIONHANDLER\_H

26.

A web page to view a great explanation on how to detect collision detection is <https://www.jeffreythompson.org/collision-detection/rect-rect.php>. The page illustrates how the algorithm works.

The algorithm is:

Is the RIGHT edge of r1 to the RIGHT of the LEFT edge of r2?

Is the LEFT edge of r1 to the LEFT of the RIGHT edge of r2?

Is the BOTTOM edge of r1 BELOW the TOP edge of r2?

Is the TOP edge of r1 ABOVE the BOTTOM edge of r2?

The image below illustrates the first case: Is the RIGHT edge of r1 to the RIGHT of the LEFT edge of r2?

A orange square with black squares

AI-generated content may be incorrect.

Figure - collision between two rectangles

It does not matter which rectangle is regarded as r1 or r2. If the large orange rectangle is regarded as r1 than the second rule for collision detection would apply. The implementation for CheckCollision():

1. bool CollisionHandler::CheckCollision(SDL\_Rect a, SDL\_Rect b){

2. bool x\_overlaps = (a.x < b.x + b.w) && (a.x + a.w > b.x);

3. bool y\_overlaps = (a.y < b.y + b.h) && (a.y + a.h > b.y);

4. return (x\_overlaps && y\_overlaps);

5. }

Line #2 covers the rules:

Is the RIGHT edge of r1 to the RIGHT of the LEFT edge of r2?

Is the LEFT edge of r1 to the LEFT of the RIGHT edge of r2?

Line #3 covers the rules:

Is the BOTTOM edge of r1 BELOW the TOP edge of r2?

Is the TOP edge of r1 ABOVE the BOTTOM edge of r2?

In our case we utilize the struct of SDL\_Rect since it captures all the information we need and is used to draw objects on the screen.

A rectangular object with text and arrows

AI-generated content may be incorrect.

Figure - SDL\_Rect struct

* Edit the Engine.h to support the Engine class being able to provide the Game Map:

1. int GameMap\* GetMap() { return m\_LevelMap;}

We will use the GetMap() method to obtain the game map. We are actually interested in the Layer named “T1” since we stated these are the tiles that define the places the player can collide with.

The GameMap maintains a vector data structure of the current game map layers:

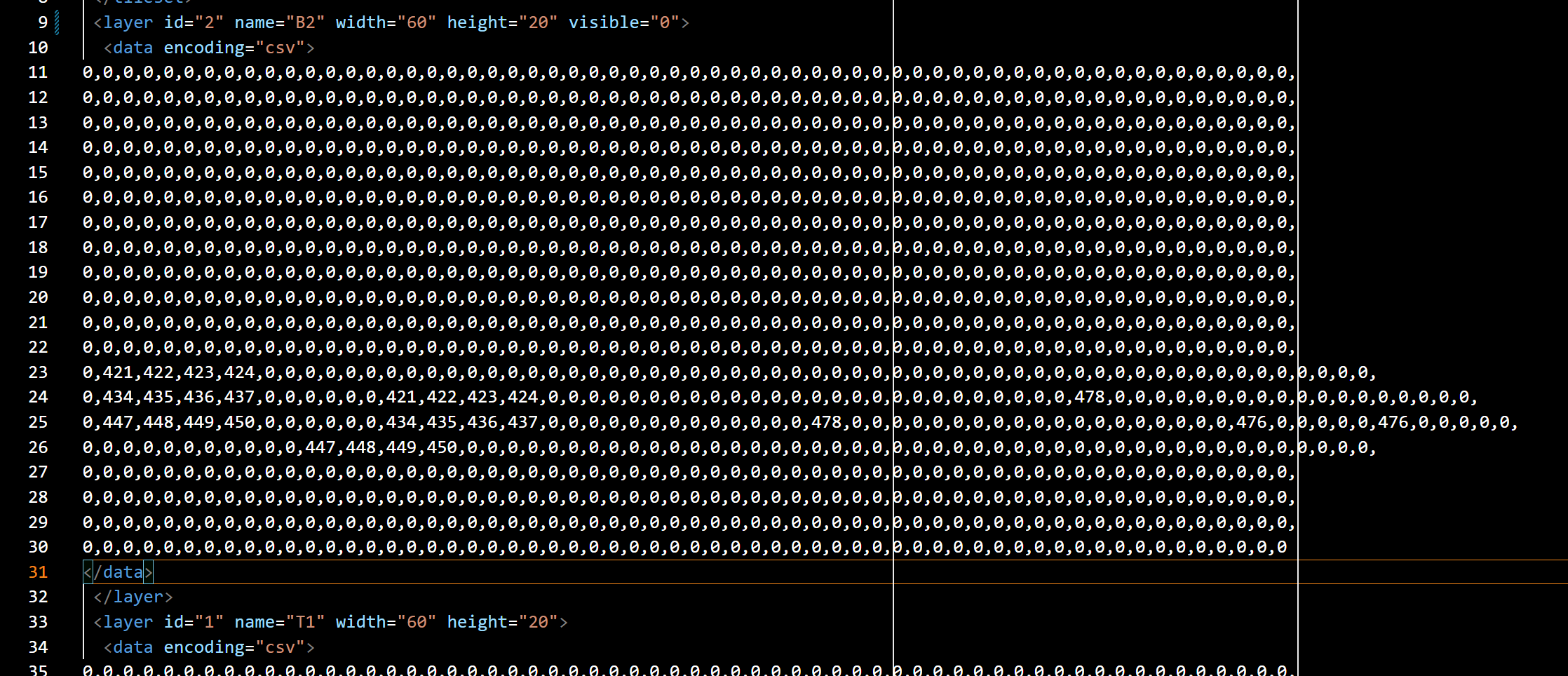
1. std::vector<Layer\*> GetMapLayers() {

2. return m\_MapLayers;

3. }

4.

We will not get this vector list and extract “T1” or the one that is to be used for detecting collisions. The code assumes that the layers T1 and B2 are processed in a certain order and added to the vector list. Examining the my\_map.tmx file, we see:



The layer named “B2” is processed first and the layer named “T1” is processed second.

If we recall the code that parsed the layers we have:

1. // iterate through all the <layer> elements

2. for(TiXmlElement\* e=root->FirstChildElement(); e!= nullptr; e=e->NextSiblingElement()){

3. if(e->Value() == std::string("layer")){

4. TileLayer\* tilelayer = ParseTileLayer(e, tilesets, tilesize, rowcount, colcount);

5. gamemap->m\_MapLayers.push\_back(tilelayer);

6. }

7. }

The above implies that the list contains [B2, T1] in that order.

The vector method back() returns the last element in a vector. So we will need to following line to extract the T1 layer:

1. m\_CollisionLayer = (TileLayer\*)Engine::GetInstance()->GetMap()->GetMapLayers().back();

Once we have the “collision” layer we can then obtain the tiled map in that layer.

Note: This is not a good way of finding and processing “collision” layers. A better way is to add a custom property to the layer, something like “collision=true” and find the layer(s) with that property! I always find when you make an assumption about the ordering of things you process you will often create maintainability issues.

* Edit the code for the CollisionHandler constructor:

1. CollisionHandler::CollisionHandler(){

2. m\_CollisionLayer = (TileLayer\*)Engine::GetInstance()->GetMap()->GetMapLayers().back();

3. m\_CollisionTilemap = m\_CollisionLayer->GetTileMap();

4. }

## Finding collision between the player and the layer tile map

The CollisionHandler::MapCollision returns true if there is a collision between the player and any tile in the game map.

To determine the tile a player character is standing on in a 2D game, you need to convert the character's world coordinates to tile coordinates. This is typically done by dividing the character's position by the tile size and using the floor or trunc function to get the integer tile indices.

Here's a more detailed breakdown:

**1. Understand the Coordinate System:**

World Coordinates:

The character's position in the game world is typically represented by floating-point numbers (e.g., (123.4, 56.7)).

Tile Coordinates:

Tile maps are organized in a grid, and each tile is represented by integer coordinates (e.g., (3, 1)).

**2. Convert World Coordinates to Tile Coordinates:**

**Step 1: Divide by Tile Size:**

Divide the character's world coordinates (x and y) by the size of a tile (e.g., 32 pixels).

tileX = playerPosition.x / tileSize

tileY = playerPosition.y / tileSize

**Step 2: Use floor() or trunc():**

Use either the floor() function (to round down) or trunc() function (to truncate) to get the integer tile indices. This ensures you're referencing a whole tile.

tileX = Math.floor(playerPosition.x / tileSize);

tileY = Math.floor(playerPosition.y / tileSize);

The code to determine if the player collided with a tile that they cannot go through:

1. bool CollisionHandler::MapCollision(SDL\_Rect a){

2. int tileSize = 32;

3. int RowCount = 20;

4. int ColCount = 60;

5.

6. int left\_tile = a.x/tileSize;

7. int right\_tile = (a.x + a.w)/tileSize;

8.

9. int top\_tile = a.y/tileSize;

10. int bottom\_tile = (a.y + a.h)/tileSize;

11.

12. if(left\_tile < 0) left\_tile = 0;

13. if(right\_tile > ColCount) right\_tile = ColCount;

14.

15. if(top\_tile < 0) top\_tile = 0;

16. if(bottom\_tile > RowCount) bottom\_tile = RowCount;

17.

18. for(int i = left\_tile; i <= right\_tile; ++i){

19. for(int j = top\_tile; j <= bottom\_tile; ++j){

20. if(m\_CollisionTilemap[j][i] > 0){

21. return true;

22. }

23. }

24. }

25.

26. return false;

27. }

Let’s see how the above works with a specific example:

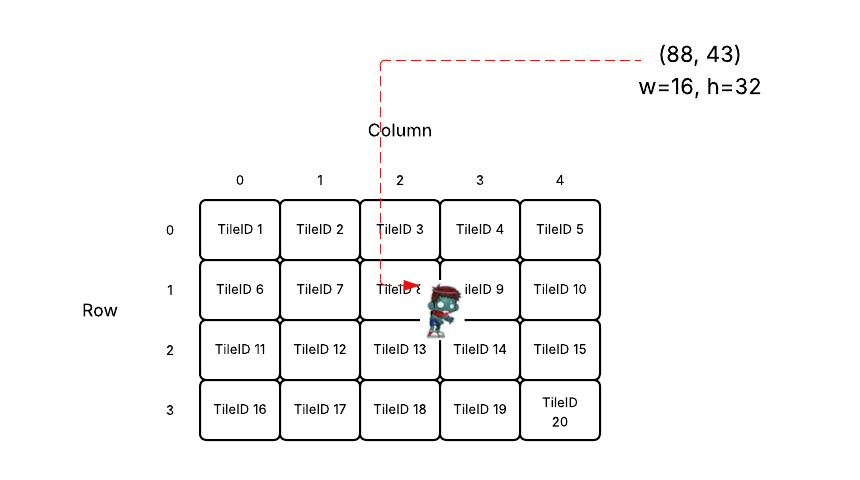


Figure - Where our player resides on the tile map

The argument SDL\_Rect a represents the rectangular area of our player. The zombie player above is at location (88, 43). The TileSize in the above example is 32 pixels in width and height). Our player has a width of 16 and height of 32 pixels.

int left\_tile = a.x/tileSize;

The above calculates the left\_tile = 88/32 = 2.75 or just 2

int right\_tile = (a.x + a.w)/tileSize;

The right\_tile = (88 + 16)/32 = 3.25 or just 3

int top\_tile = a.y/tileSize;

We find the player must be between columns of 2 and 3:

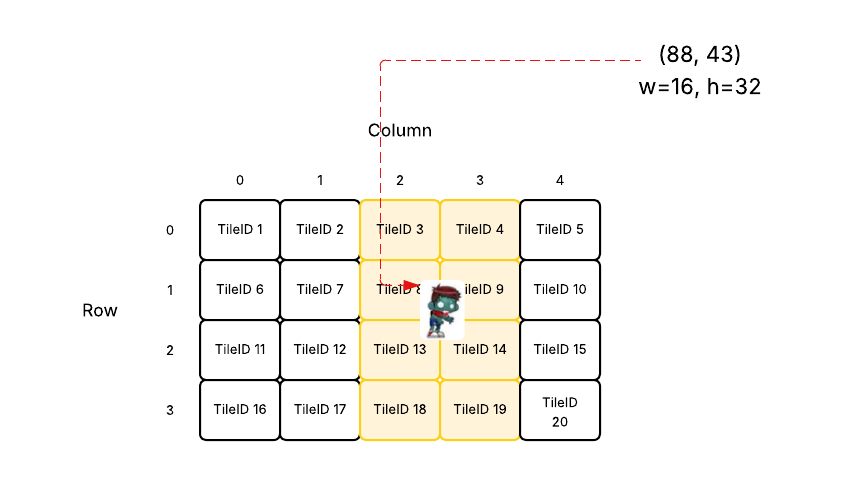


Figure - Player hits tiles in columns 2 and 3

top\_tile = 43 / 32 = 1.34 or 1

int bottom\_tile = (a.y + a.h)/tileSize;

bottom\_tile = (43 + 32) / 32 = 64 / 32 = 2.3 = 2

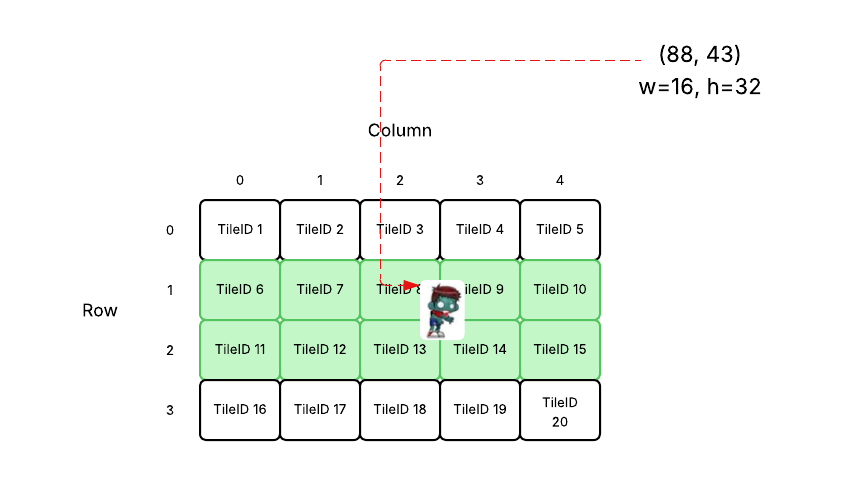


Figure - Player is on tile rows 1 and 2

None of the values are out of bounds (off the map), so the next thing is to iterate through the tiles bounded by the left\_tile and right\_tile value (columns) and through the tiles that bound the top and bottom-most tiles (the rows). If the player is within a tile that not equal to 0 then it must be a “collision” area.

18. for(int i = left\_tile; i <= right\_tile; ++i){

19. for(int j = top\_tile; j <= bottom\_tile; ++j){

20. if(m\_CollisionTilemap[j][i] > 0){

21. return true;

22. }

23. }

24. }

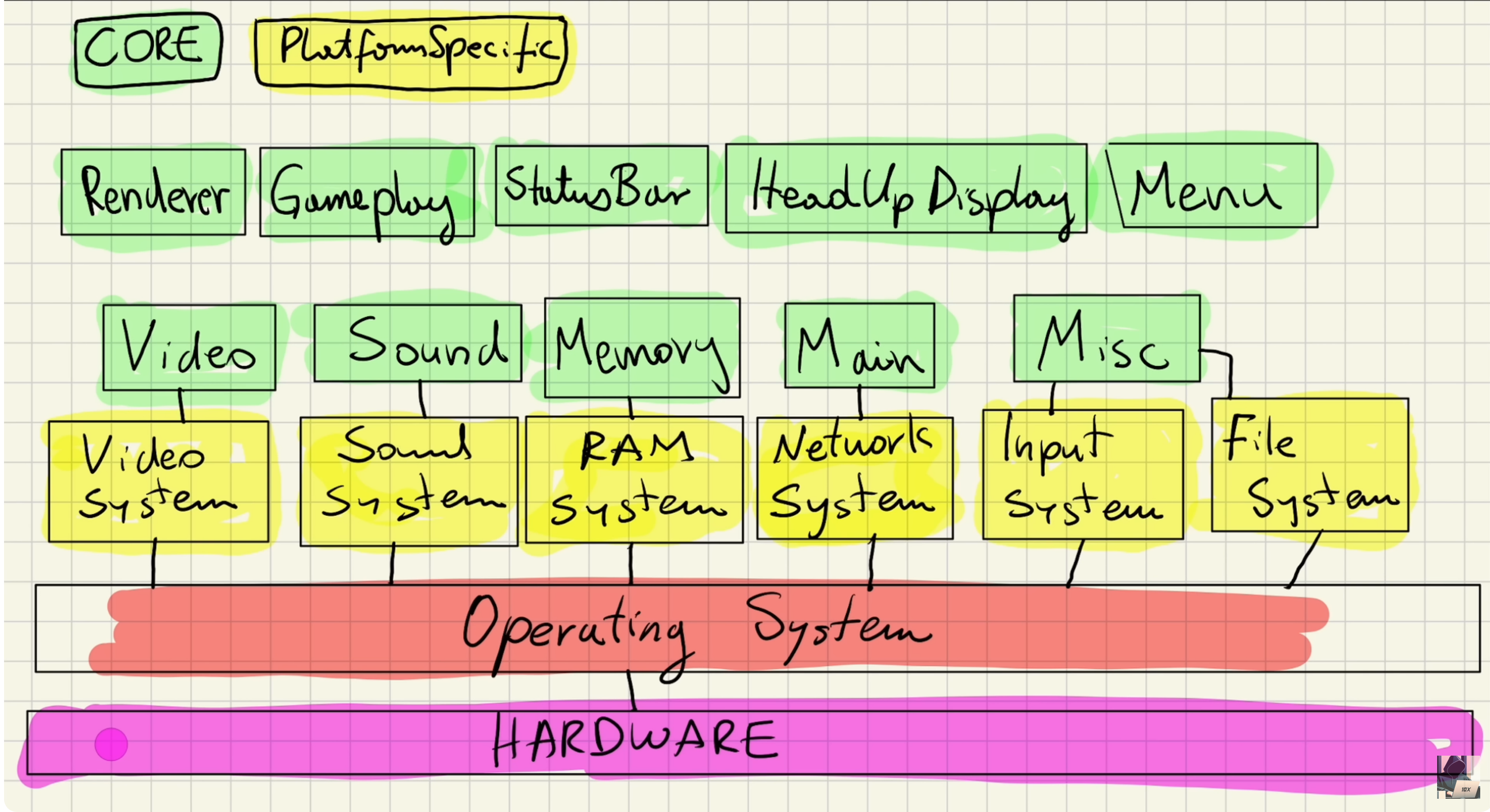
# 20. Collision Handling – Part II

# 21. Animation State Machine

# 22. Texture Parser

# 23. Frame Animation

# 24. Enemy Animation



From: <https://www.youtube.com/watch?v=cqL3jvlU61c&ab_channel=Tariq10x>

# Web sites to Learn SDL2

* <https://lazyfoo.net/tutorials/SDL/index.php>
* <https://wiki.libsdl.org/SDL2/FrontPage>
* <https://wiki.libsdl.org/SDL2_image>
* <https://www.ferzkopp.net/wordpress/2016/01/02/sdl_gfx-sdl2_gfx/>
* <https://www.freepik.com/>
* <http://programarcadegames.com/>
* <https://giphy.com/>
* <https://github.com/nsklaus/SoftEngine>
* <https://forum.gdevelop.io/t/solved-how-do-i-slice-a-sprite-sheet/37755>
* <https://box2d.org/>
* <https://www.gameart2d.com/> - Tile sets

**Syntax**

**Function Parameters**

**Returns**

**Remarks**