TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

Computer Graphics Project for Fourth Semester

PROJECT: Shoot 'Em

Team Members:

Avinash Aryal (076BEI009)
Diwas Adhikari (076BEI014)
Kshitiz Pandey(076BEI019)
Parit Pokharel (076BEI021)

Submitted To:

Department of Electronics and Computer Engineering,
Pulchowk Campus

March, 2022

PREFACE

ACKNOWLEDGEMENT:

First of all, we thank our lecturer, Lok Nath Regmi for giving us an opportunity to deliver an implementation to the concepts we analyzed and studied in our Computer Graphics courseware.

We tried our best to implement different algorithms for designing a simple scroller game with this project. We tried to collaborate and discuss among the teammates for the best results and we are pretty happy with what we have accomplished throughout the journey of this project. These projects help us enhance our understanding related to computer science and channelize our logical thinking ability.

INTRODUCTION

Shoot Em' !

2D – Scroller Game

Shoot Em' ! (*Gotta Kill Em' All)* is a graphical context created to represent a two-dimensional camera movement. It is a program in which we can control our own character in a way that we desire with keyboard input such that it moves across a map that is larger than that particular graphical context. The project is a composite of different mechanics that create a two-dimensional game.

Here, the objective is to basically infiltrate the Norse village and eliminate all dwarves. By 'shoot', we mean that the character can create energy orbs to kill the raged dwarves. The camera follows the player trying to keep the player as the center of attention for the perspective of gameplay. As the player moves, the camera traverses the map or say, the map offsets along with the player to give that illusion.

OBJECTIVES:

The objectives of this project are as follows:

1. To help understand graphics intuitively since, users are allowed to get entertained while all the complexities of graphical framework and rendering remains hidden making the concept of graph becomes easier to understand for them.

2. To make it easier for everyone to understand different graphical algorithms incorporating it into layers using different colors and textures making the experience visually appealing.

3. To collaborate as a team and work together.

4. To create a graphical interface in which user can grasp the actions easily and of course, HAVE FUN !!!!!

INTRODUCTION TO GRAPHICS FRAMEWORK

As, our objective (previously on 'Objective' section), was to create a graphical framework that is certainly able to render 2D-graphics on a window screen so that, we will be able to implement it for various graphical programs i.e. games, simulations, graphs, etc.

Since, we have generalized it as a framework, it is a skeleton of graphical rendering operations that is just able enough to start a window on its own which can be fused with textures, music and text on that rendering screen.

2.1. Approach to the framework:

Before making this project, we have been certain about the thing that it is not a graphical engine that can be manipulated by either CUI or, GUI interfaces. But, it will just be a main class that gets assisted by its helper classes. (not to be confused with: sub-classes or, inherited classes)

Basically, we are going to create a main class (let's say: Launcher) that would have graphic rendering functions which can be used to render graphics intertwined with other features, generated in other helper classes (let's say: Texture, Music, etc.) for respective purposes.

2.2. Requirements for the framework:

Though, we have decent knowledge of programming using C++ language, we still are unaware of the graphical abilities of the operating system. In Windows, the system uses Win32 API for the purpose of graphical features and functions. But, it's extremely tedious to learn it since it is way more hardware-oriented. Hence, we could also have tried OpenGL as a graphics library but, we were not used to code in it and also, we also didn't require projections, 3-D rendering, shaders, etc. for rendering display. Hence, we picked up SDL2.0 as our graphics library to make this simple 2-D graphics framework.

2.3. Why and How SDL2.0 is used ?

Simply, SDL2.0 is preferred as our library for graphical display because it's cross-platform, easy to use and completely free. It is way easier to create a graphical context in SDL2.0 than any other library (Except: SFML). Moreover, it is also capable of using OpenGL, Vulkan, etc. for 3-D rendering. (though, we don't require it.)

Our general plan in this project was to mask the functions of SDL2.0 in separate classes under separate namespaces so that, SDL2.0 will act as an underneath graphic driver under our framework. Our multiple classes in the framework can simply distribute the functionalities of SDL2.0 as separate entities referred by separate classes which could help us preserve the OOP architecture in programming. We haven't let go of the OOP techniques which includes encapsulation and data abstraction for the sake of our project. We have taken OOP and graphics rendering of SDL2.0 side by side, to generate a working framework to code graphics.

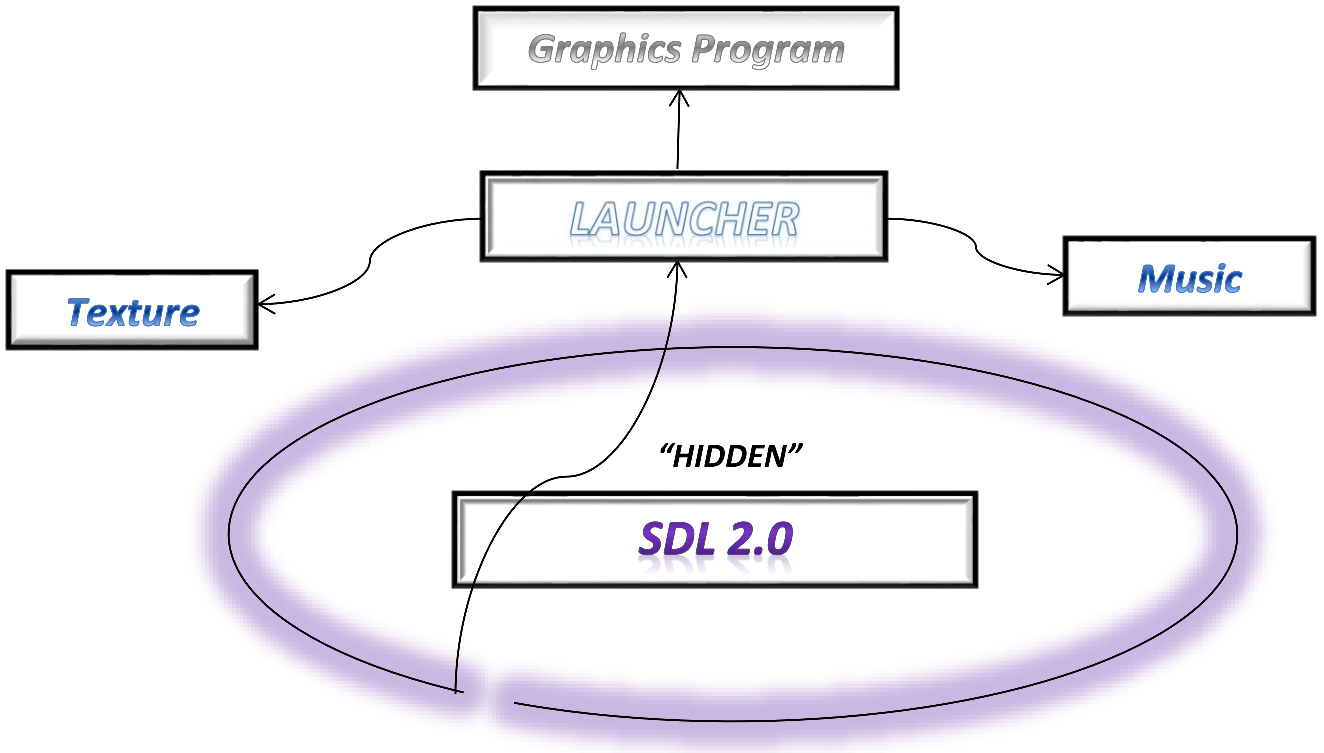


Fig: SDL2.0 adding graphics layer underneath the framework

2.4. UML Diagram Representation:

Launcher is basically an object to produce a graphical context that will be assisted by the helper classes which, in case of our project are: Texture, Font, Music and Timer. It would grab the functionalities from each of those classes by addressing scopes for the particular class in the main class which is, of course Launcher. The thing not to be confused about this project is that, it is a wrapper class or a framework for the ease of coding 2-D graphics for the programmer but, not exactly the user. The software that gets build out of this class would be the final product for the user.

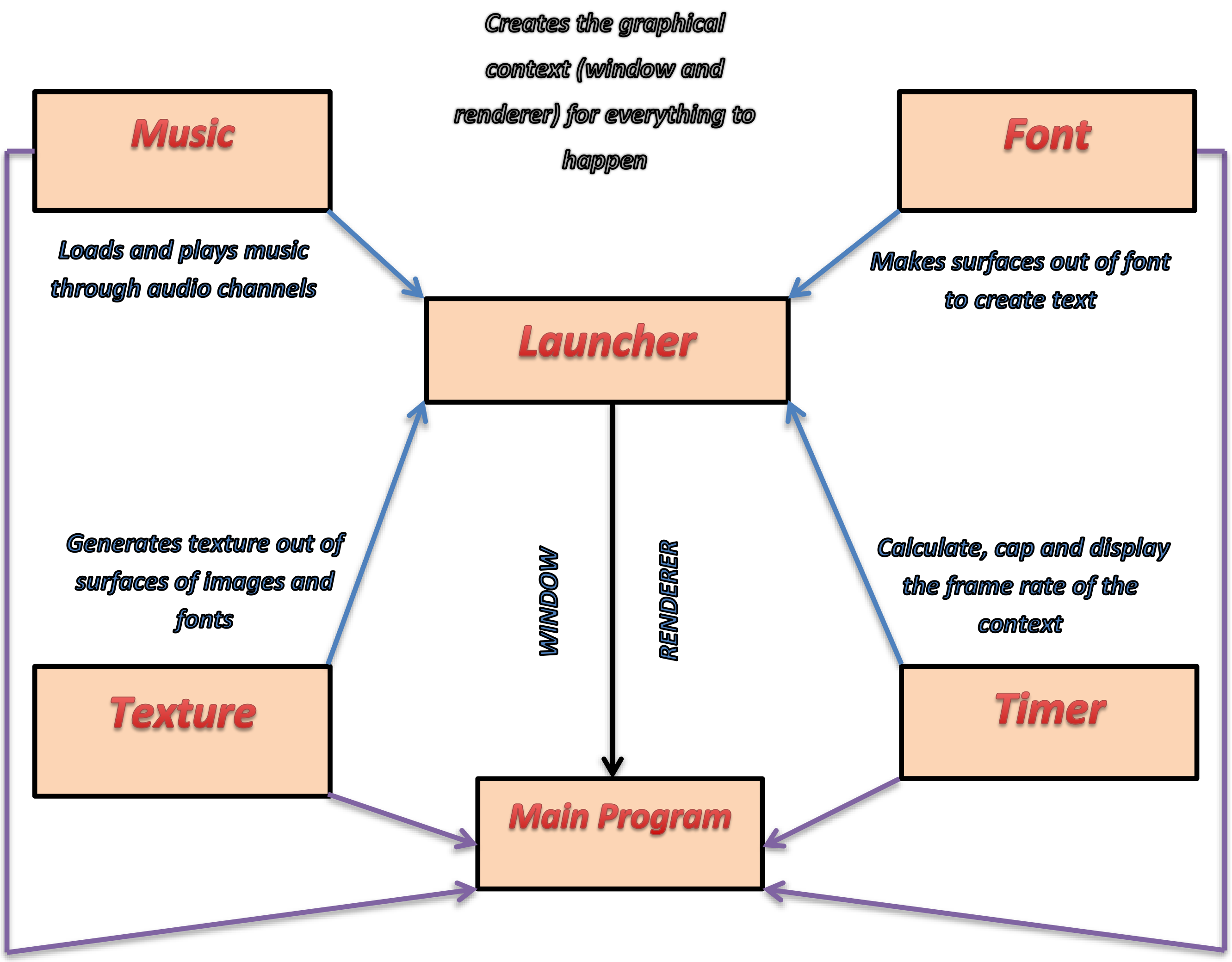
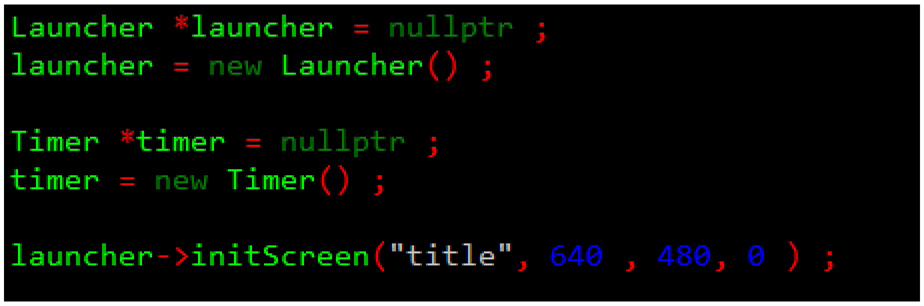

Creates the graphical context (window and renderer) for everything to happen
Music
Font
Loads and plays music through audio channels
Makes surfaces out of font to create text
Launcher
Generates texture out of surfaces of images and fonts
WINDOW
RENDERER
Calculate, cap and display the frame rate of the context
Texture
Timer
Main Program

Fig: UML Diagram for the Framework

2.5. Making a sample program with the framework:

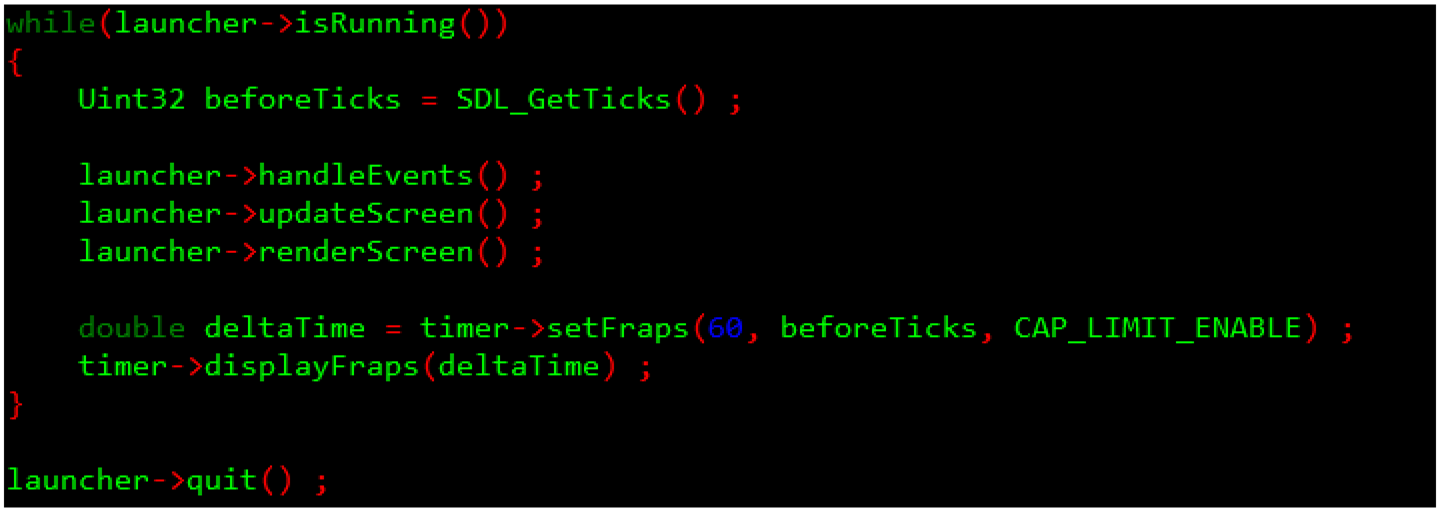
There is no better way to check if this program works than making a program out of it. So, we created a sample program which creates a simple window on the center of the screen with the dimension which we desire.

We created a source file to include Launcher class which is of course, the main class which holds all the other classes together. Then, in the main() function, we created Launcher and Timer object for their respective purposes in the program. We initialized the screen with required dimensions and title using initScreen() function keeping the fullscreen mode disabled. This automatically, initializes SDL2.0 as well as creates window and renderer for the program.



```
Launcher *launcher = nullptr ;
launcher = new Launcher() ;

Timer *timer = nullptr ;
timer = new Timer() ;

launcher->initScreen("title", 640 , 480, 0 ) ;
```

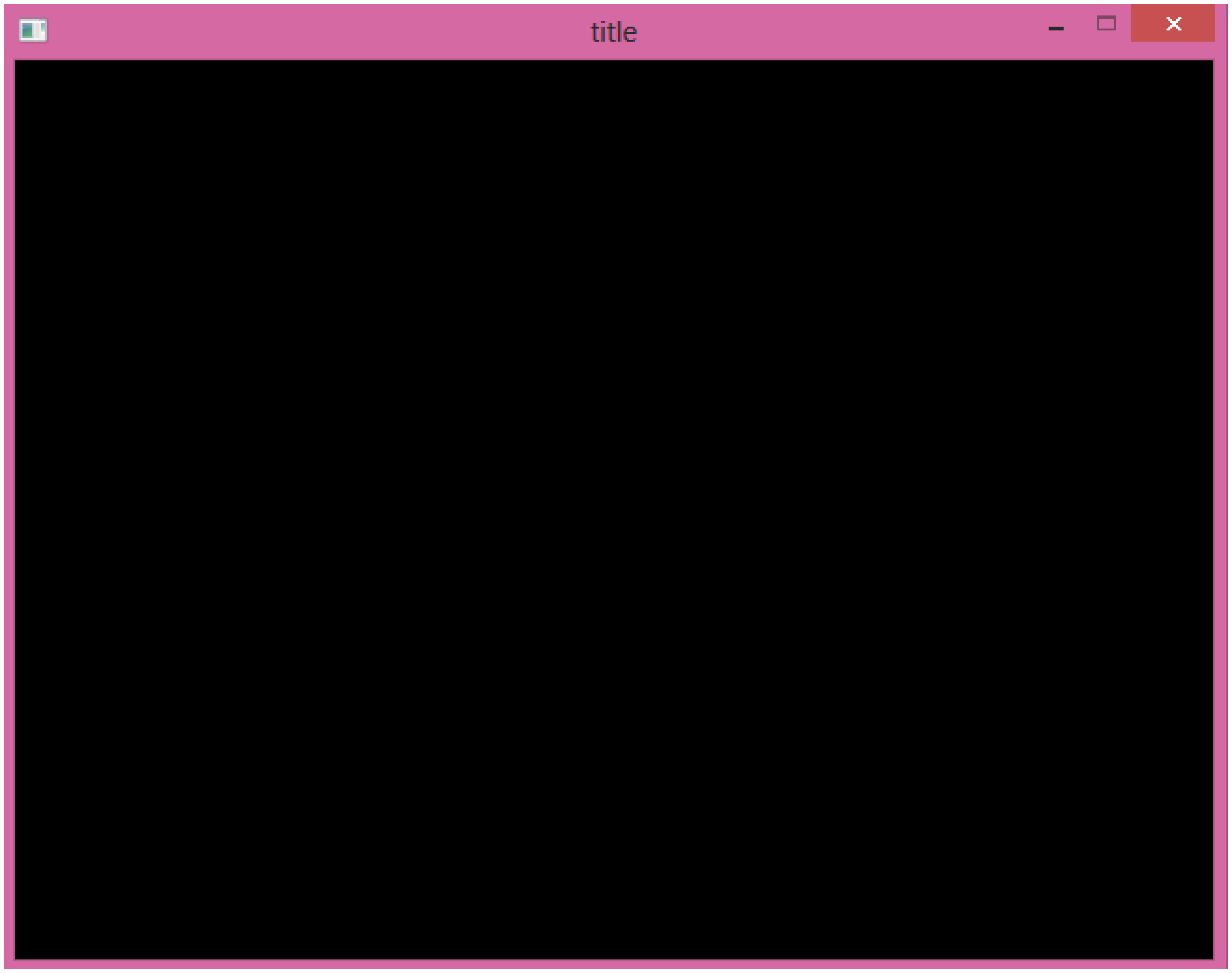
While initializing the screen, we already set a flag that allows generating frames continuously for the renderer till, the screen is quitted. Thus, within the frame, we handled our window events and even cleared the renderer.



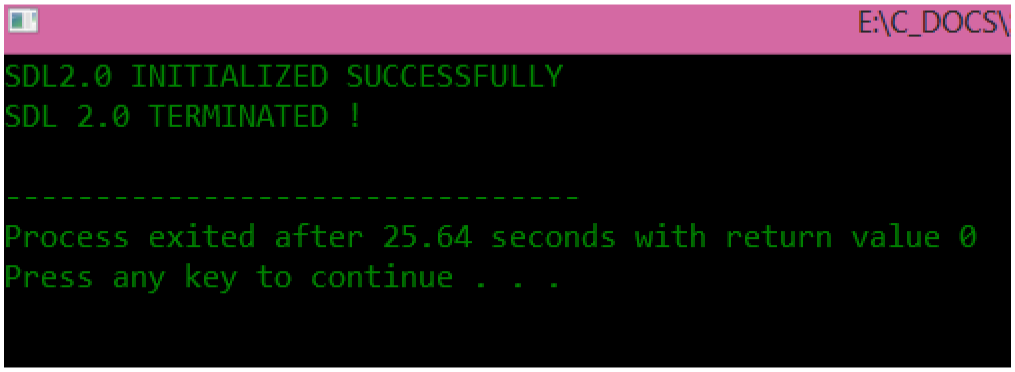
```
while(launcher->isRunning())
{
    Uint32 beforeTicks = SDL_GetTicks() ;

    launcher->handleEvents() ;
    launcher->updateScreen() ;
    launcher->renderScreen() ;

    double deltaTime = timer->setFraps(60, beforeTicks, CAP_LIMIT_ENABLE) ;
    timer->displayFraps(deltaTime) ;
}

launcher->quit() ;
```

Timer was just used to check if it’s capping the frame rate to 60Hz (60 frames per second). If Launcher is quitted from events linked to I/O or display, Launcher will disable the running flag to quit the window and renderer.

We witnessed the window of 640*480 dimensions with title “title” as per the arguments provided in the code. Also, the screen was cleared with black color and presented on the screen.



Thus, the console window in this case would display dialogs declared in the Launcher class while executing the program. If ‘ESC’ key or quit button is pressed, it displays message referring to the termination of SDL2.0 library.

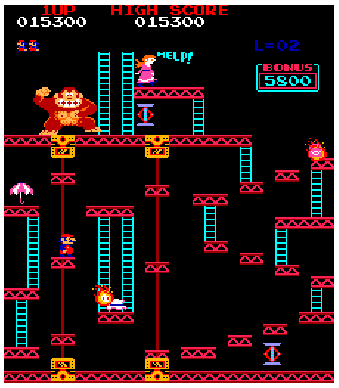


```
SDL2.0 INITIALIZED SUCCESSFULLY
SDL 2.0 TERMINATED !


--------------------------------
Process exited after 25.64 seconds with return value 0
Press any key to continue . . .
```

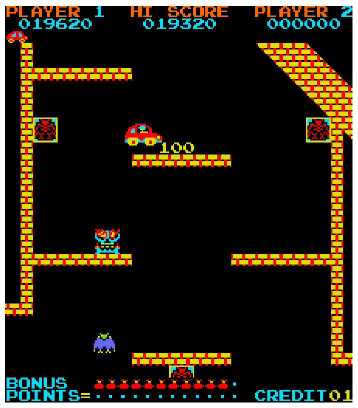
HISTORY OF 2-D GAMES:

2D platform games are a variation of platform video games that were very popular genre of video games. 2D platformers originated in the early 1980s and in the mid 1990s, platformers made the transition to 3D.

Because of the technical limitations back in the days, early examples were confined to a static playing field, generally viewed in profile and were based on climbing rather than jumping. Donkey Kong, an arcade game created by Nintendo and released in July 1981, was the first game to allow players to jump over obstacles and gaps; it is widely considered to be the first platformer due to these defining features. The following year, Donkey Kong received a sequel, Donkey Kong Jr. and later Mario Bros., a platform game with two-player cooperative play. It laid the groundwork for other two-player cooperative platformers such as Fairyland Story and Bubble Bobble. Beginning in 1982, transitional games emerged that did not use scrolling graphics, but had levels that span several connected screens.

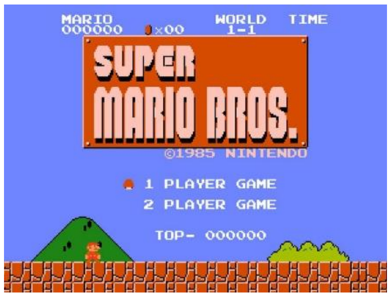


The first platform game to use scrolling graphics came years before the genre became popular. Jump Bug is a platform-shooter developed by Alpha Denshi under contract for Hoei/Coreland and released to arcades in 1981, only five months after Donkey Kong. Players control a bouncing car that jumps on various platforms such as buildings, clouds, and hills. Jump Bug offered a glimpse of what was to come, with uneven,



suspended platforms and levels that scrolled horizontally and, in one section, vertically. Irem's 1982 arcade game Moon Patrol combines jumping over obstacles and shooting attackers. A month later, Taito released Jungle King, a side-scrolling action game with parallax scrolling and some platform elements: jumping between vines, jumping or running beneath bouncing boulders.

Nintendo's Super Mario Bros., released for the Nintendo Entertainment System in 1985, became the archetype for many platform games. It was bundled with Nintendo systems in North America, Japan, and Europe, and sold over 40 million copies, according to the 1999 "Guinness Book of World Records". Its success as a pack-in led many companies to see platform games as vital to their success, and contributed greatly to popularizing the genre during the 8-bit console generation.



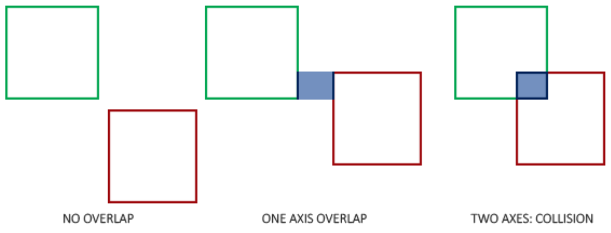
In the 1970s, most vertically scrolling games involved driving. The first vertically scrolling video game was Taito's *Speed Race*, released in November 1974. Atari's *Hi-way* was released eleven months later in 1975. Rapidly there were driving games that combined vertical, horizontal, and even diagonal scrolling, making the vertical-only distinction less important.

ALGORITHM TERMINOLOGY

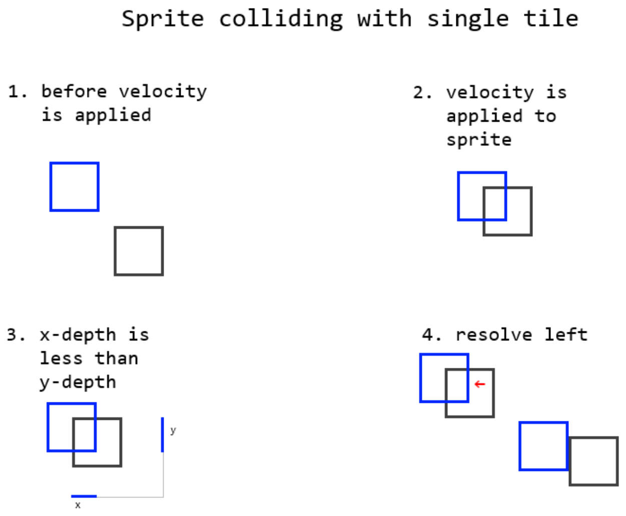
COLLISION DETECTION & RESOLUTION

Collision Detection is the computational problem of detecting the intersection of two or more objects. Collision detection is a classic issue of computational geometry and has applications in various computing fields, primarily in computer graphics, computer games, computer simulations, robotics and computational physics. Collision detection algorithm can be divided into operating on 2D and 3D objects.

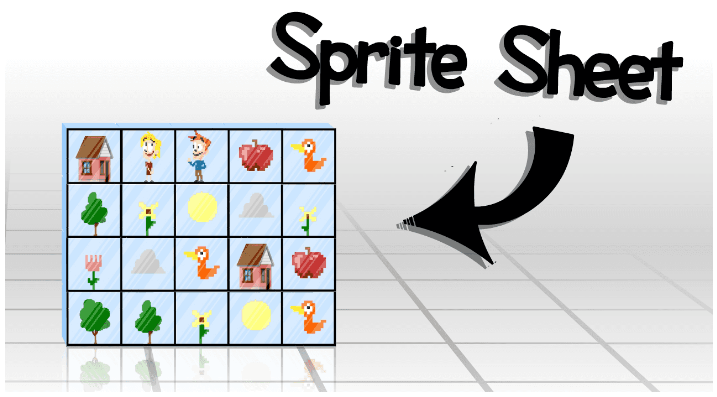
AABB stands for axis-aligned bounding box, a rectangular collision shape aligned to the base axes of the scene, which in 2D aligns to the x and y axis. Being axis-aligned means the rectangular box has no rotation and its edges are parallel to the base axes of the scene (e.g. left and right edge are parallel to the y axis). The fact that these boxes are always aligned to the axes of the scene makes calculations easier.



A collision occurs when two collision shapes enter each other's region e.g. the shape that determines the first object is in some way inside the shape of the second object. For AABBs this is quite easy to determine due to the fact that they are aligned to the scene's axes: we check for each axis if the two object's edges on that axis overlap. So we check if the horizontal edges overlap, and if the vertical edges overlap of both objects. If both the horizontal and vertical edges overlap, we have a collision.


Sprite colliding with single tile
1. before velocity is applied
2. velocity is applied to sprite
3. x-depth is less than y-depth
y
x
4. resolve left

SPRITE SHEET & ANIMATION



A sprite sheet is an image that consists of several smaller images (sprites) and/or animations. Combining the small images in one big image improves the game performance, reduces the memory usage and speeds up the startup and loading time of the game.

To use a sprite sheet, you load the sprite sheet as a single large image, and then you load the individual images from the sprite sheet image. This turns out to be much more efficient than loading a bunch of separate image files.

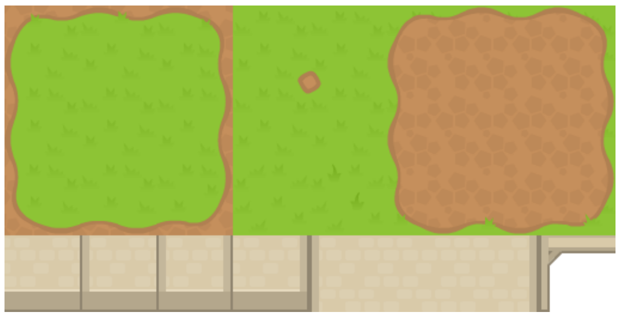
TILING

Tilemaps are a very popular technique in 2D game development, consisting of building the game world or level map out of small, regular-shaped images called tiles. This results in performance and memory usage gains — big image files containing entire level maps are not needed, as they are constructed by small images or image fragments multiple times.

The most efficient way to store the tile images is in an atlas or spritesheet. This is all of the required tiles grouped together in a single image file. When it's time to draw a tile, only a small section of this bigger image is rendered on the game canvas.

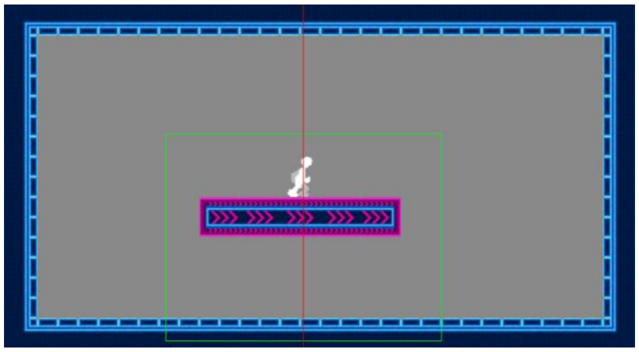
Square-based tilemaps are the most simple implementation. A more generic case would be rectangular-based tilemaps — instead of square — but they are far less common. Square tiles allow for two perspectives:

- Top-down (like many RPG's or strategy games like Warcraft 2 or Final Fantasy's world view.)
- Side-view (like platformers such as Super Mario Bros.)

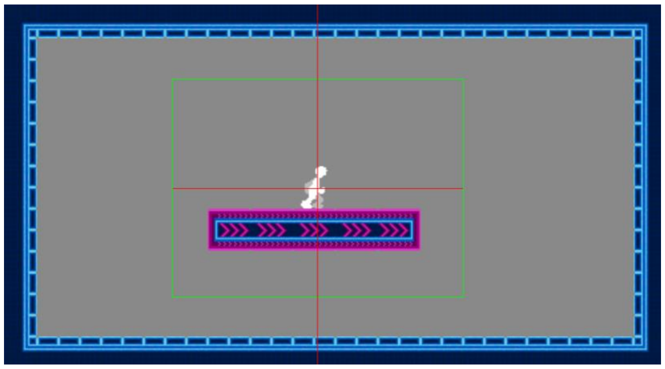


CAMERA MOVEMENT

In all scrolling games, we need a translation between world coordinates (the position where sprites or other elements are located in the level or game world) and screen coordinates (the actual position where those elements are rendered on the screen). The world coordinates can be expressed in terms of tile position (row and column of the map) or in pixels across the map, depending on the game. To be able to transform world coordinates into screen coordinates, we need the coordinates of the camera, since they determine which section of the world is being displayed.



Position tracking camera is a bit more involved: the viewport tracks the position of the player and moves accordingly, so to keep the character centered on the screen. There are two types of position tracking cameras that are used in videogames: horizontal-tracking and full-tracking cameras.



Horizontal-tracking cameras keep the player in the center of the screen horizontally, while jumps don’t influence the camera position. This is ideal for games that span horizontally, since we won’t have the camera moving when jumping and temporarily hiding enemies we may fall on.

Sometimes our levels don’t span only horizontally, so we need to track the player in both axes, keeping it in the center of the screen at all times. This is full-tracking camera. This is good for platformers that don’t require extremely precise maneuvering, since precise maneuvering could result in way too much movement from the camera.

RENDERING

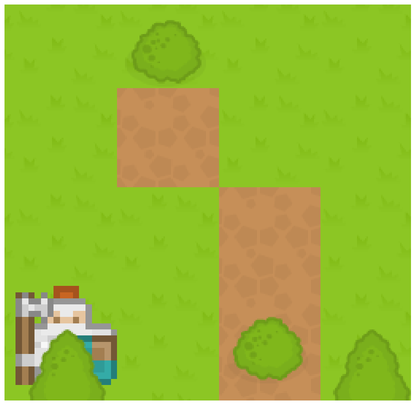
A trivial method for rendering would just be to iterate over all the tiles (like in static tilemaps) and draw them, subtracting the camera coordinates and letting the parts that fall outside the view window sit there, hidden. Drawing all the tiles that cannot be seen is wasteful, however, and can take a toll on performance. Only tiles that are at visible should be rendered ideally.

LAYERS

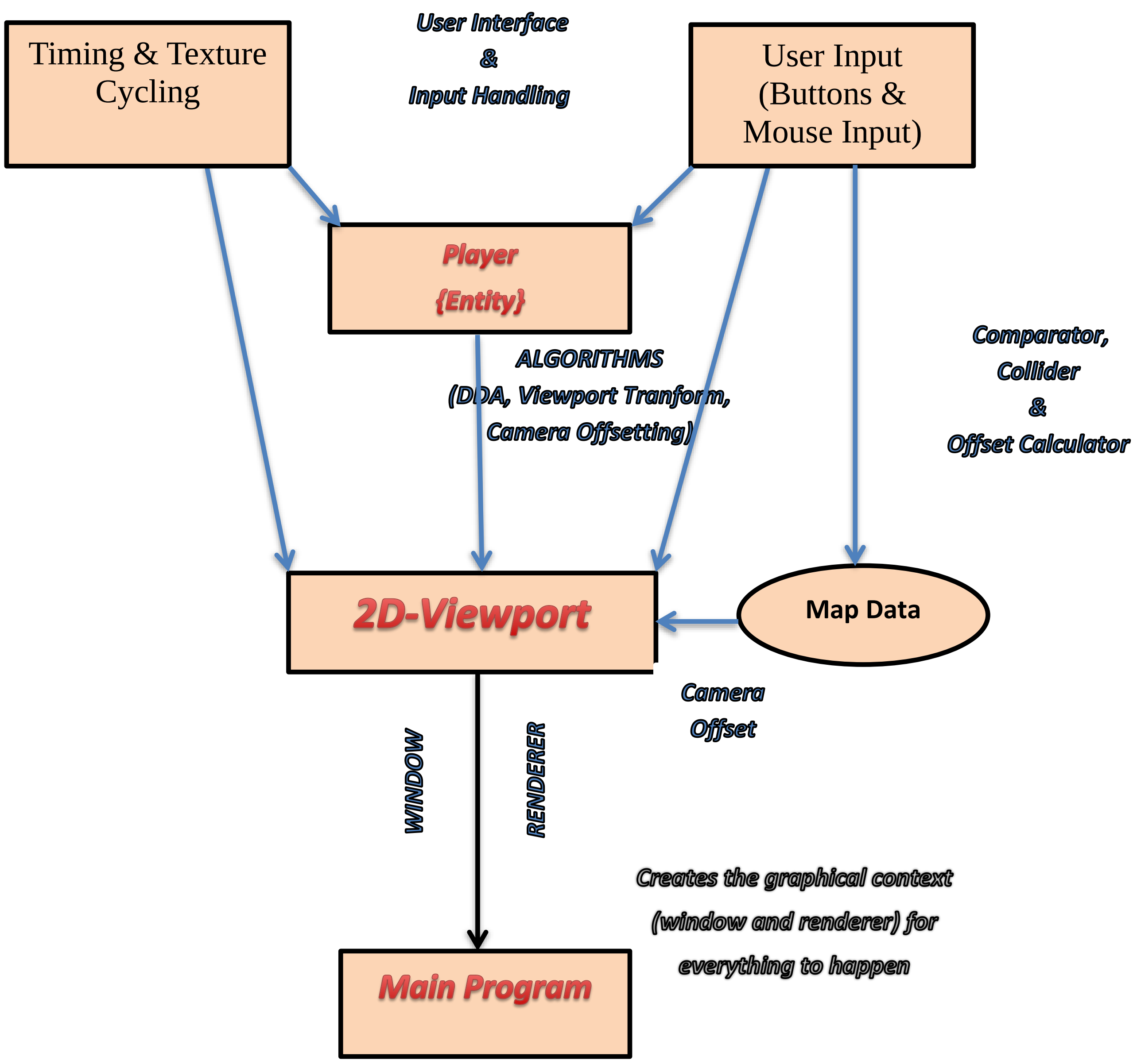
The visual grid is often made up of several layers. This allows us to have a richer game world with fewer tiles, since the same image can be used with different backgrounds. For instance, a rock that could appear on top of several terrain types (like grass, sand or brick) could be included on its own separate tile which is then rendered on a new layer, instead of several rock tiles, each with a different background terrain.

If characters or other game sprites are drawn in the middle of the layer stack, this allows for interesting effects such as having characters walking behind trees or buildings.

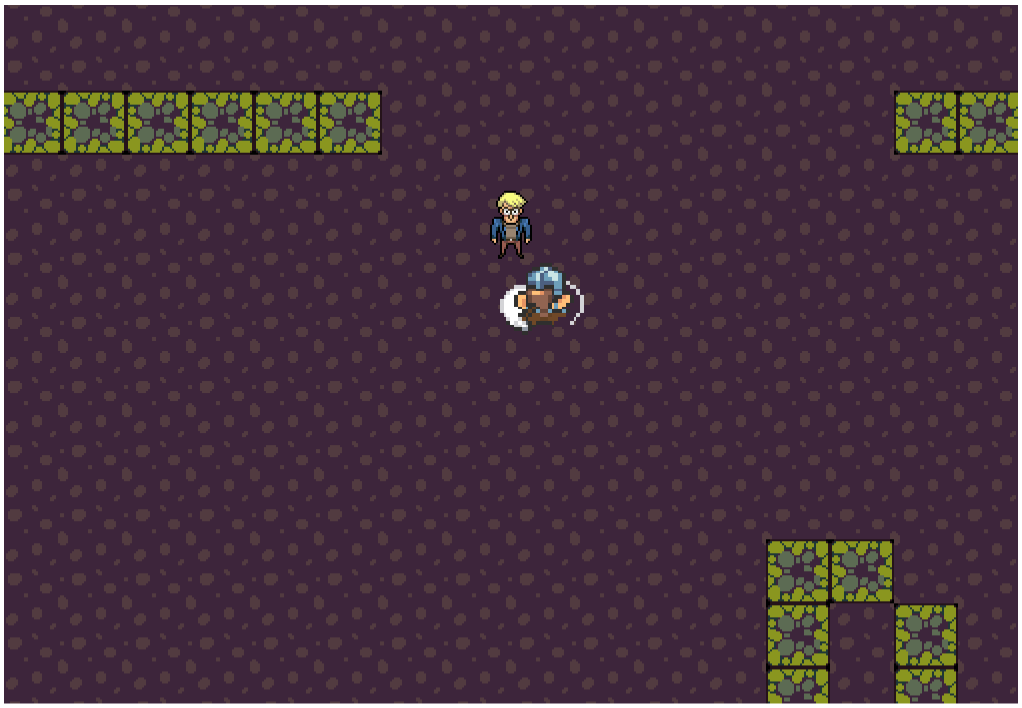
The figure shows an example of both points: a character appearing behind a tile (the knight appearing behind the top of a tree) and a tile (the bush) being rendered over different terrain types.

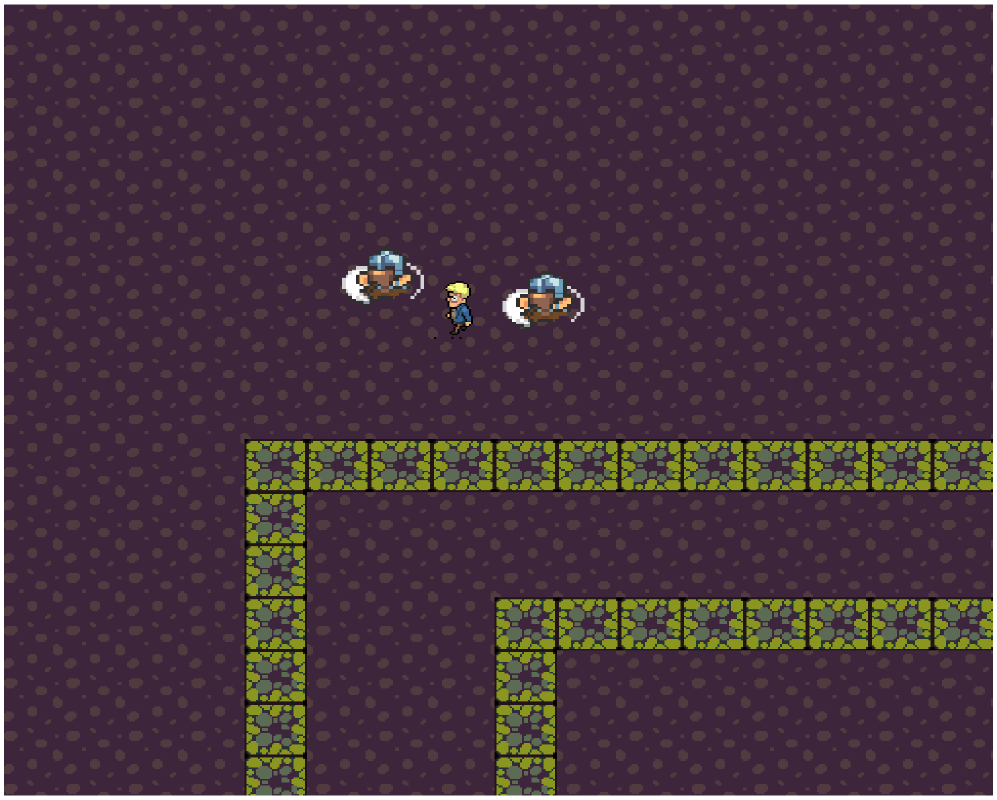


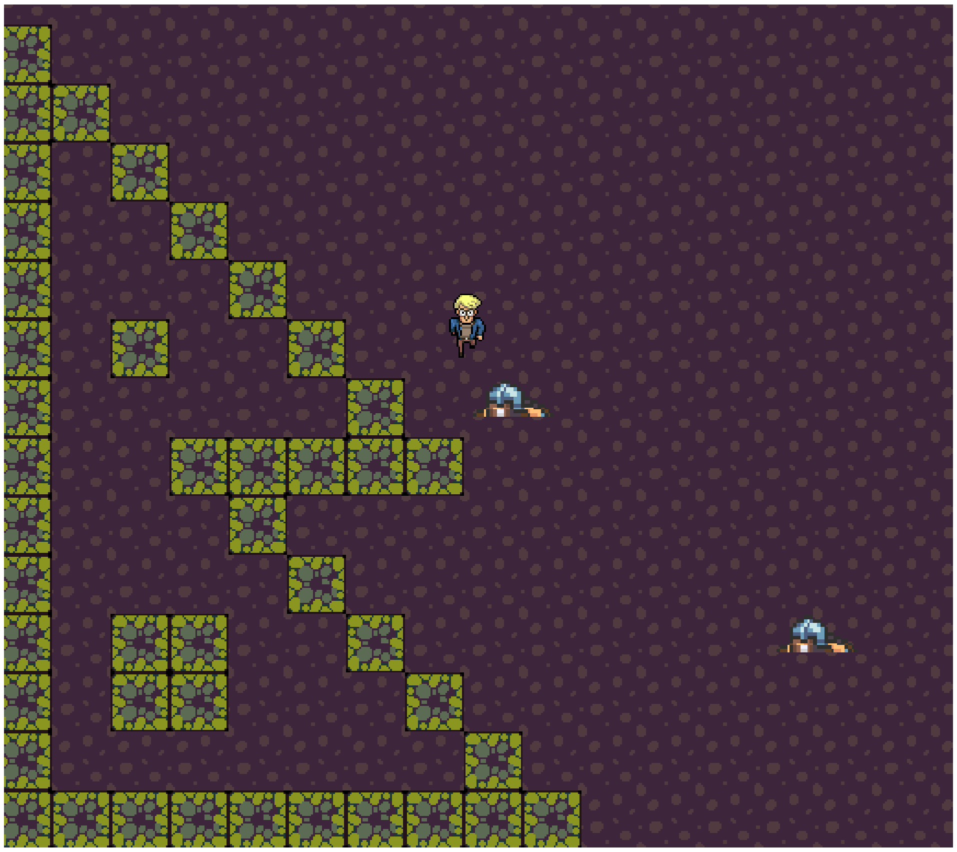
SYSTEM BLOCK DIAGRAM

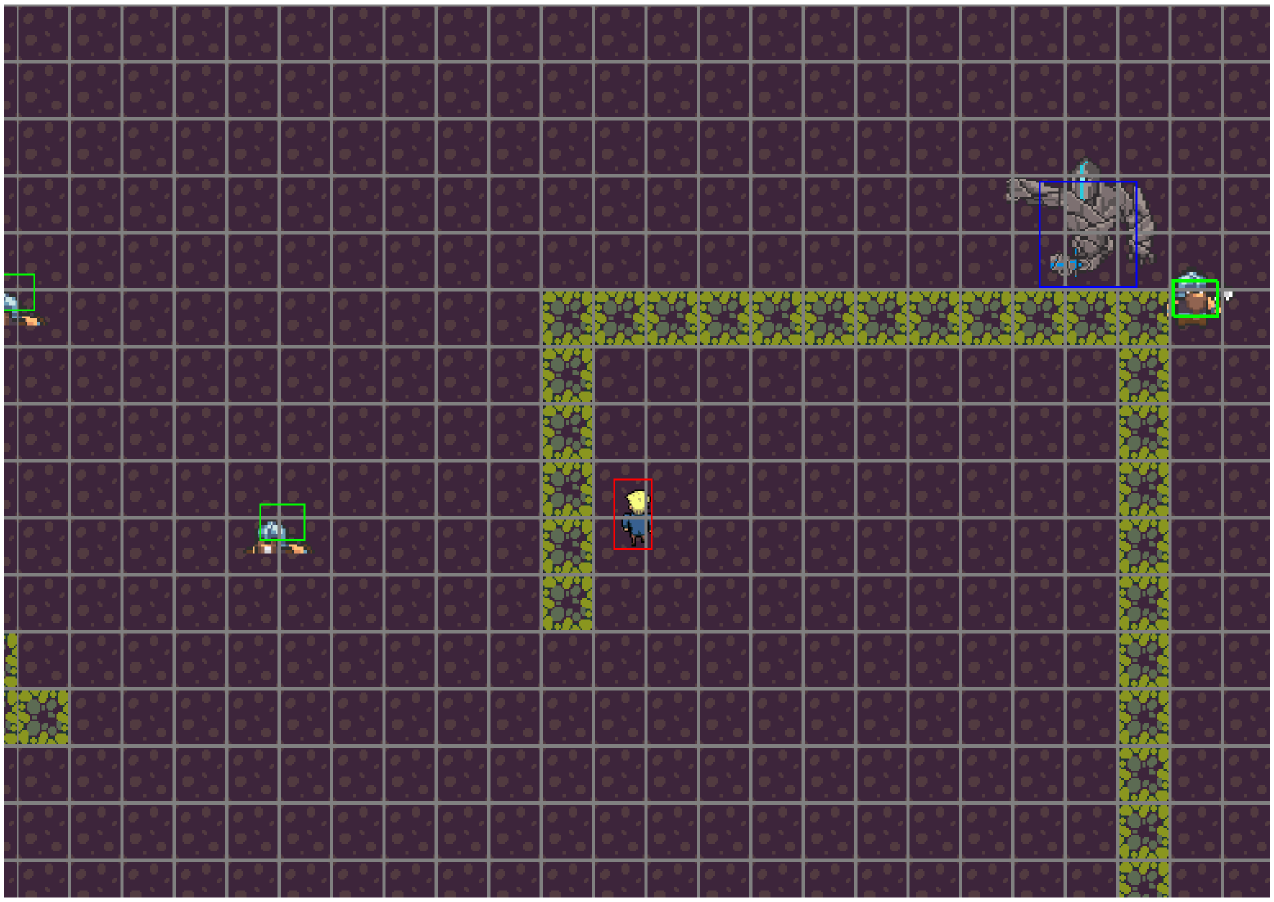

Timing & Texture Cycling
User Interface & Input Handling
User Input (Buttons & Mouse Input)
Player {Entity}
ALGORITHMS (DDA, Viewport Tranform, Camera Offsetting)
Comparator, Collider & Offset Calculator
2D-Viewport
Map Data
Camera Offset
WINDOW
RENDERER
Creates the graphical context (window and renderer) for everything to happen
Main Program

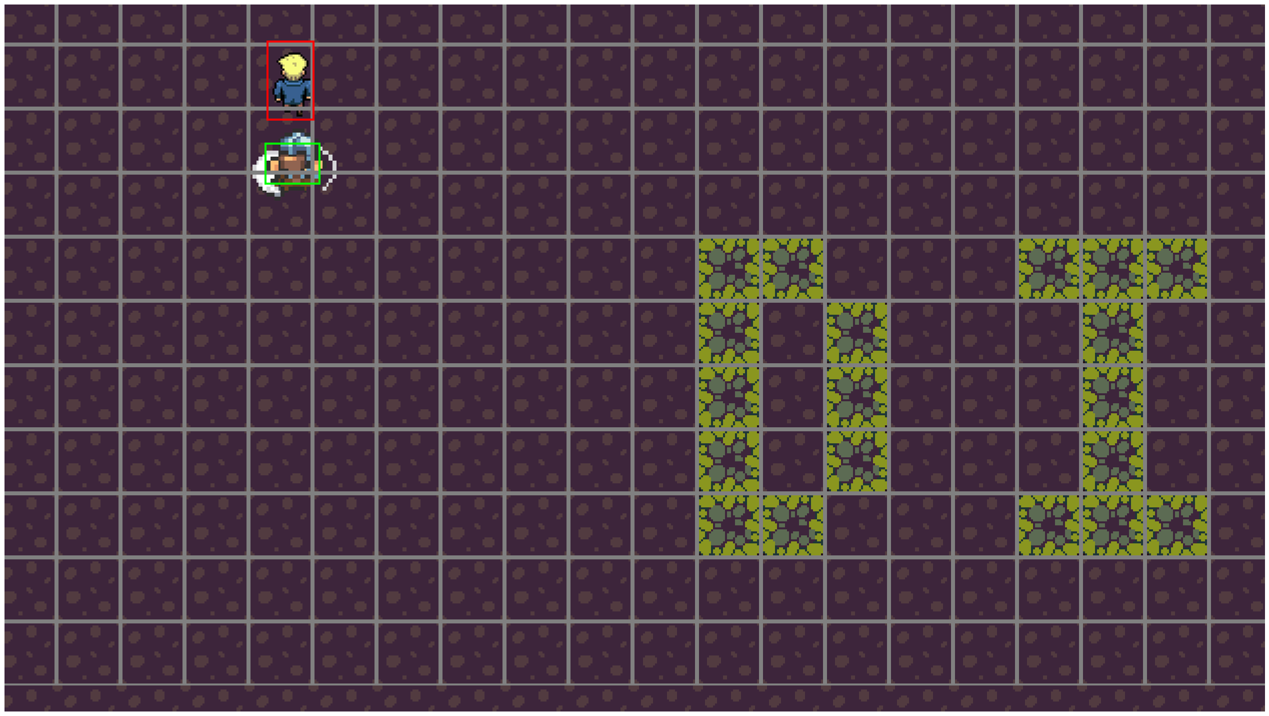
OUTPUT ANALYSIS











IMPROVEMENTS TO MAKE

1. To make a proper UI for health points (HP), damage points (DP), inventory and even a reference map.

2. To use a parser such as Tiny XML parser to load resources and map data.

3. To make a proper level design since, we only implemented the underlying structure but, it is not worth being a fun game yet.

4. To create a uniform timer for same performances in different devices by frame limit capping and generate proper instances of events during level progression.

5. To implement a path finding algorithm such as A* algorithm to improvise enemy AI as the enemy AI is limited to differential analyzer that has a limited linearized functionality that doesn't incorporate collision.
(This single-handedly improvises the gameplay.)