Project Phase 2: Methodology Date: 27- 07- 2024

Ahmad Fares, Nour Saneh, AbedRahman Mesto.

Game development in 2024:

The Criteria in the ease of indie game development today is how efficient and friendly a certain engine for visual programing, to begin we will cross reference this Github repository :

https://github.com/JohnDuncanScott/2d-game-engine-comparison

with the different games our team has made in the past, and with the popular game Minecraft that was originally made using java.

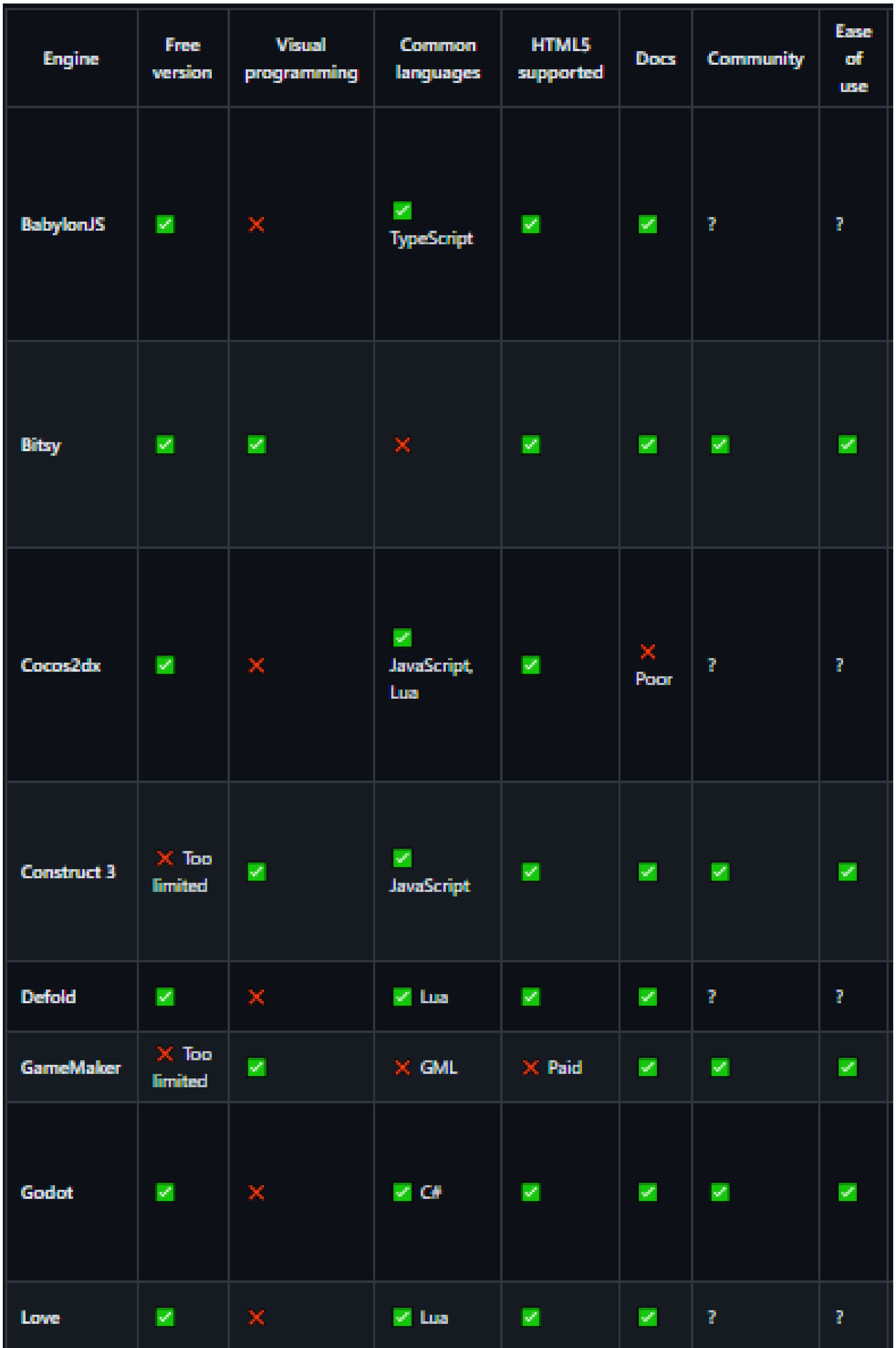
And how each engine is unique based on the language it uses, the big O when it comes to game iterations, and how well it handles garbage collection and smart rendering so that it may be compatible with 60fps on an 8gb ram machine with a decent graphics card.

From the github article:

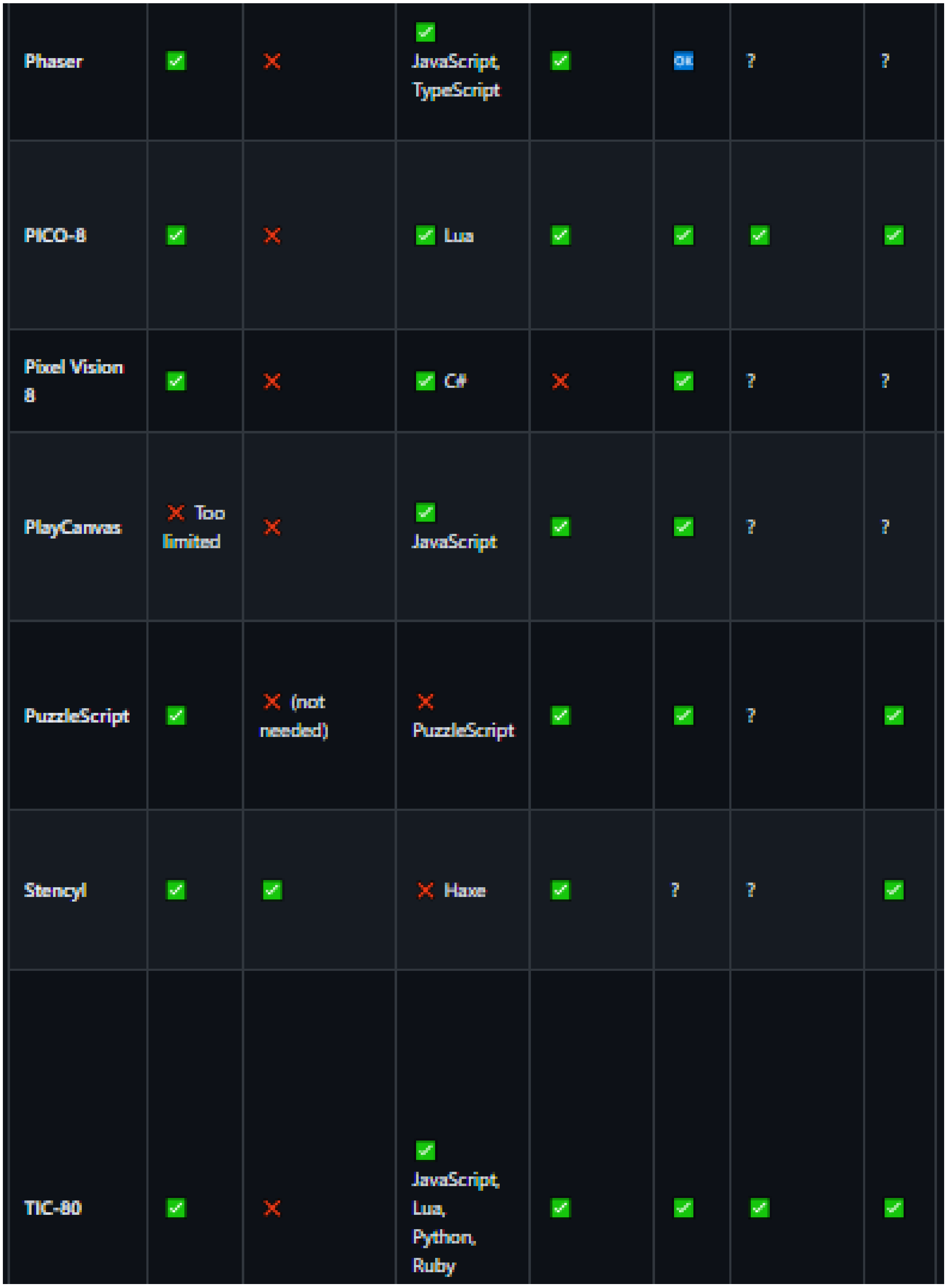
Language guide

This project only focuses on engines that offer visual programming (in addition to common languages). However, you may wish to start with visual programming and then dig into a popular programming language (e.g. for job purposes). Choosing a language is ultimately a personal preference. Below is a brief guide for the common ones:

- JavaScript - popular in web development
- TypeScript - typed version of JavaScript. Gaining popularity in web development
- Java / C# - solid all-rounders. Syntax is similar. Usually seen in backend services. C# is popular in games too (e.g. **Unity** engine, **Godot** engine)
- Python - easy to read. Popular in AI / machine learning. Good as a general purpose scripting language for automation
- Lua - easy to embed in other applications. Easy to learn. Popular in games (e.g. **Love** engine, Roblox, World of Warcraft interface customisation, etc.)
- C++ - subjectively harder than the others to learn. Great for optimisation. Usually seen in high performance systems, low level software and games (e.g. **Unreal Engine**)



Engine	Free version	Visual programming	Common languages	HTML5 supported	Docs	Community	Ease of use
BabylonJS	✅	❌	✅ TypeScript	✅	✅	?	?
Bitsy	✅	✅	❌	✅	✅	✅	✅
Cocos2dx	✅	❌	✅ JavaScript, Lua	✅	❌ Poor	?	?
Construct 3	❌ Too limited	✅	✅ JavaScript	✅	✅	✅	✅
Defold	✅	❌	✅ Lua	✅	✅	?	?
GameMaker	❌ Too limited	✅	❌ GML	❌ Paid	✅	✅	✅
Godot	✅	❌	✅ C#	✅	✅	✅	✅
Love	✅	❌	✅ Lua	✅	✅	?	?



Phaser	✅	❌	✅ JavaScript, TypeScript	✅	🆗	?	?
PICO-8	✅	❌	✅ Lua	✅	✅	✅	✅
Pixel Vision 8	✅	❌	✅ C#	❌	✅	?	?
PlayCanvas	❌ Too limited	❌	✅ JavaScript	✅	✅	?	?
PuzzleScript	✅	❌ (not needed)	❌ PuzzleScript	✅	✅	?	✅
Stencyl	✅	✅	❌ Haxe	✅	?	?	✅
TIC-80	✅	❌	✅ JavaScript, Lua, Python, Ruby	✅	✅	✅	✅

The article shows how some engines work better then others but there are cases where only using visual studios and the right set of tools will build you a decent game without having to go into game engines and you would be setting the stage to make your own game engine such as stardew valley and minecraft.

First we will be comparing how python vs java differs when it comes to game dev using our own games from the past.

1.python (soduko solver)

What are the characteristics of python?

1- Python's syntax is designed to be readable and straightforward, making it simple and easy to learn among beginners.

2- Python is an interpreted language, which means it is executed line by line without the need for compilation into machine code. This allows for quicker development cycles as developers can immediately see the result of their code.

3- In python variables are dynamically typed, meaning you don't have to declare the date type of a variable before using it. This feature allows for more readability.

4- Python is a versatile programming language meaning it can be used in a wide variety of applications, including web development, automation, data science, artificial intelligence.

5- Python supports multiple programming paradigms, including object-oriented, procedural, and functional programming. This allows developers to choose the most suitable approach for their projects.

Python stands out from other programming languages due to its emphasis on readability and simplicity in syntax, its interpreted nature which eliminates the need for compilation, its dynamic typing system that doesn't require explicit variable declarations, its versatility across a wide range of applications, and its support for multiple programming paradigms including object-oriented, procedural, and functional programming. The characteristics make python a popular choice for developers especially those in the field of data science, artificial intelligence and automation.

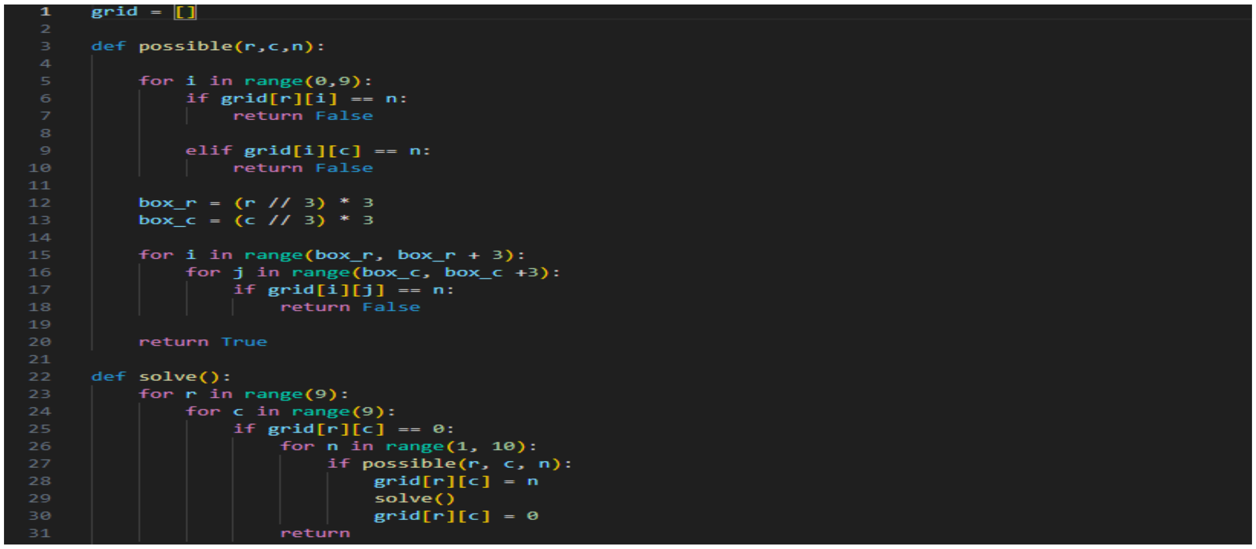
How is python utilized?

1- Python is utilized in the game development business for activities like tool development, prototyping, and game programming. Many features of well-known video games, such as Civilization IV and Battlefield 2, have been developed with Python participation.

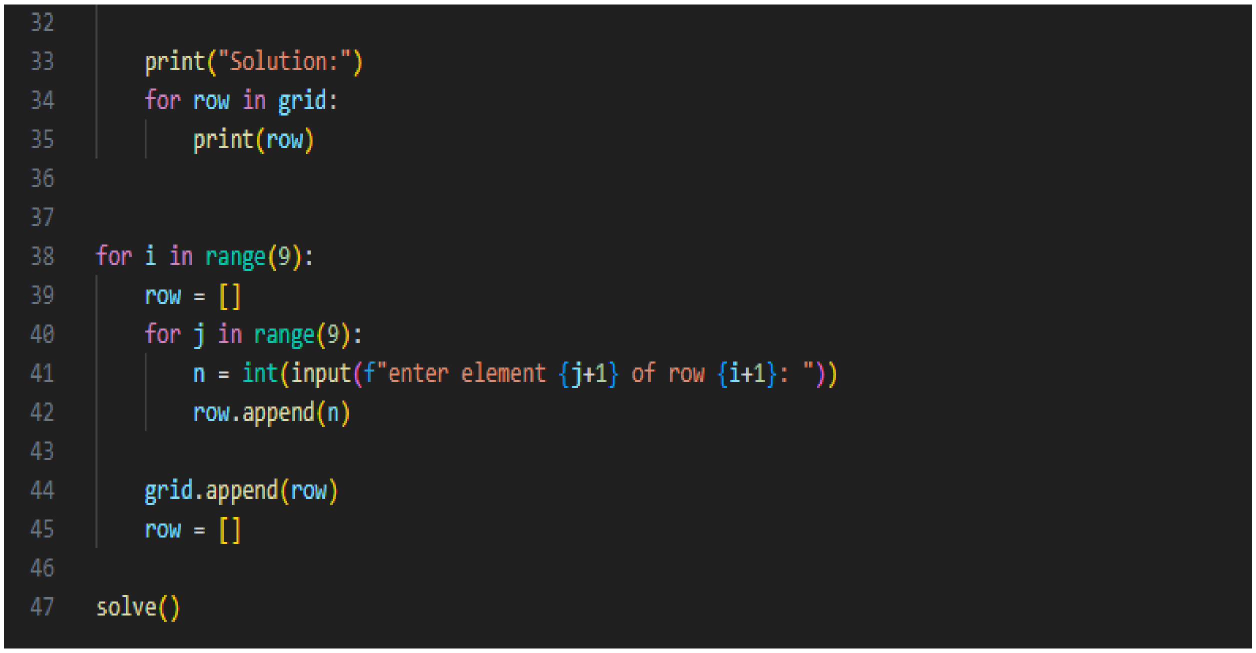
2- Automation and Scripting: Python is a popular choice for system administration, scripting, and automation activities because to its ease of use and readability. It is employed in the creation of utilities, server management, and repetitive work automation.

3- Desktop GUI Applications: Python is used for creating desktop graphical user interfaces (GUIs), especially when combined with libraries such as Tkinter, PyQt, and wxPython. Programs such as BitTorrent and Dropbox.

Example on automation by making a sudoku solver



```python
grid = []

def possible(r,c,n):

    for i in range(0,9):
        if grid[r][i] == n:
            return False

        elif grid[i][c] == n:
            return False

    box_r = (r // 3) * 3
    box_c = (c // 3) * 3

    for i in range(box_r, box_r + 3):
        for j in range(box_c, box_c +3):
            if grid[i][j] == n:
                return False

    return True

def solve():
    for r in range(9):
        for c in range(9):
            if grid[r][c] == 0:
                for n in range(1, 10):
                    if possible(r, c, n):
                        grid[r][c] = n
                        solve()
                        grid[r][c] = 0
                return
```



```python

    print("Solution:")
    for row in grid:
        print(row)


for i in range(9):
    row = []
    for j in range(9):
        n = int(input(f"enter element {j+1} of row {i+1}: "))
        row.append(n)

    grid.append(row)
    row = []

solve()
```

Is python statically typed or dynamically typed?

Python uses dynamic typing. This implies that rather than at compile time, variable types are decided during runtime. Python does not need you to specify variable types explicitly; instead, they are inferred from the values that are assigned to them. For instance, you won't have any problems assigning a string value to the same variable after first assigning an integer value to it.

Object-oriented programming (OOP) is fully supported in Python. It enables the implementation of ideas like inheritance, encapsulation, and polymorphism as well as the definition of classes and objects.

Python is an interpreted language. The Python interpreter reads and runs the code line by line when you run a Python script. Different from languages like C or C++, there is no separate compilation stage. Rather, the Python interpreter executes intermediate bytecode that has been translated from the original code. Since you can see the results of your code right away without having to compile it first, this simplifies development and debugging.

Part 2: Java snake game

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

-Linus Torvalds (developer of Linux System)

Understanding the structure of the syntax and semantics can help us decrease the big Oh which means less work for the compiler, hence our program can do more with less code.

This will also make the program updatable, think of it as building a foundation of a building and going 1 level at a time, rather than mixing abstract elements in a functional way.

The first method will provide efficient code, while the second is guaranteed to have bugs.

The following is an example of how to create the classic snake game using the native java library, with 3 simple classes.

Note: we will not be using any foreign libraries such as libGDX, The libraries used are the following:

javax.Swing : GUI toolkit for java with multiple components(buttons, dialog boxes, etc..) used for crossplatform development due to efficiency.

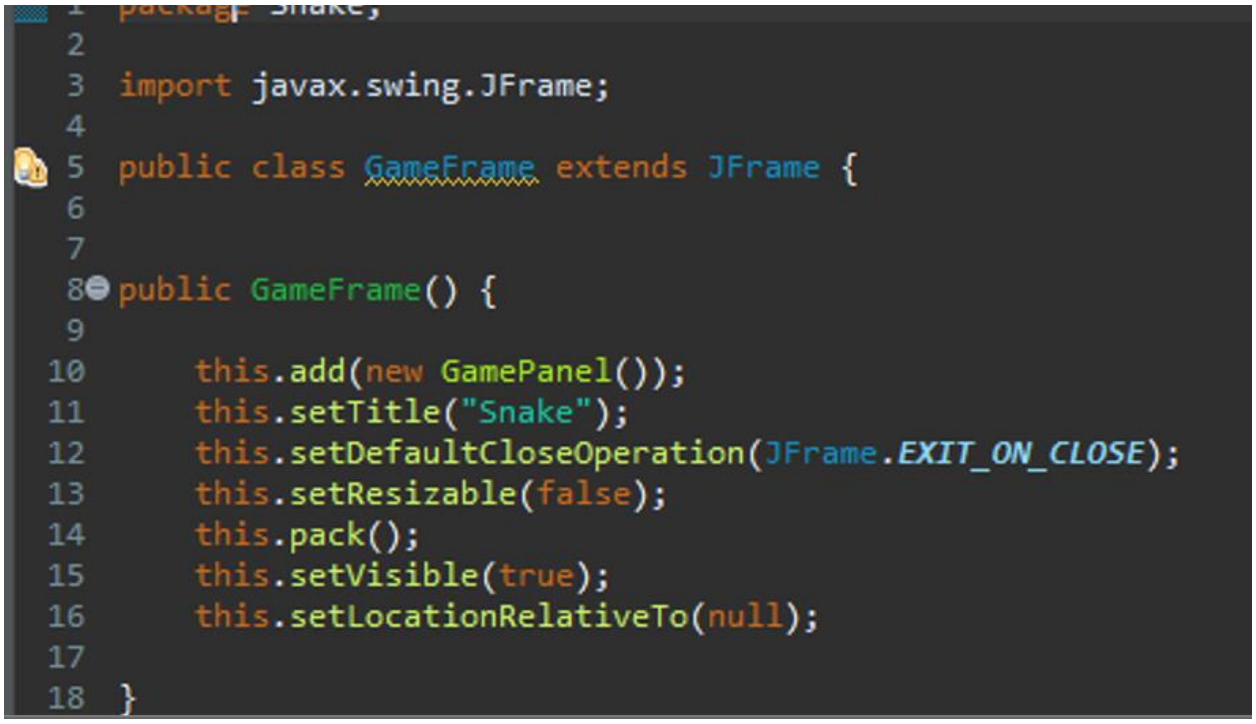
-We will only be using the Jpanel and the Jframe classes from this library since our game is simple and consists of 2 states. Running = true/false. The game will be built in the panel class which is then passed on the frame class to be displayed on screen using the main class.(main class calls Jframe which calls Jpanel).

Java.awt & Java.awt.event: The awt library is used with the swing library to obtain dynamic application with real time changes, the java.awt.event classes handle the user input and the Java.awt classes commit runtime changes to the program based on the input events.

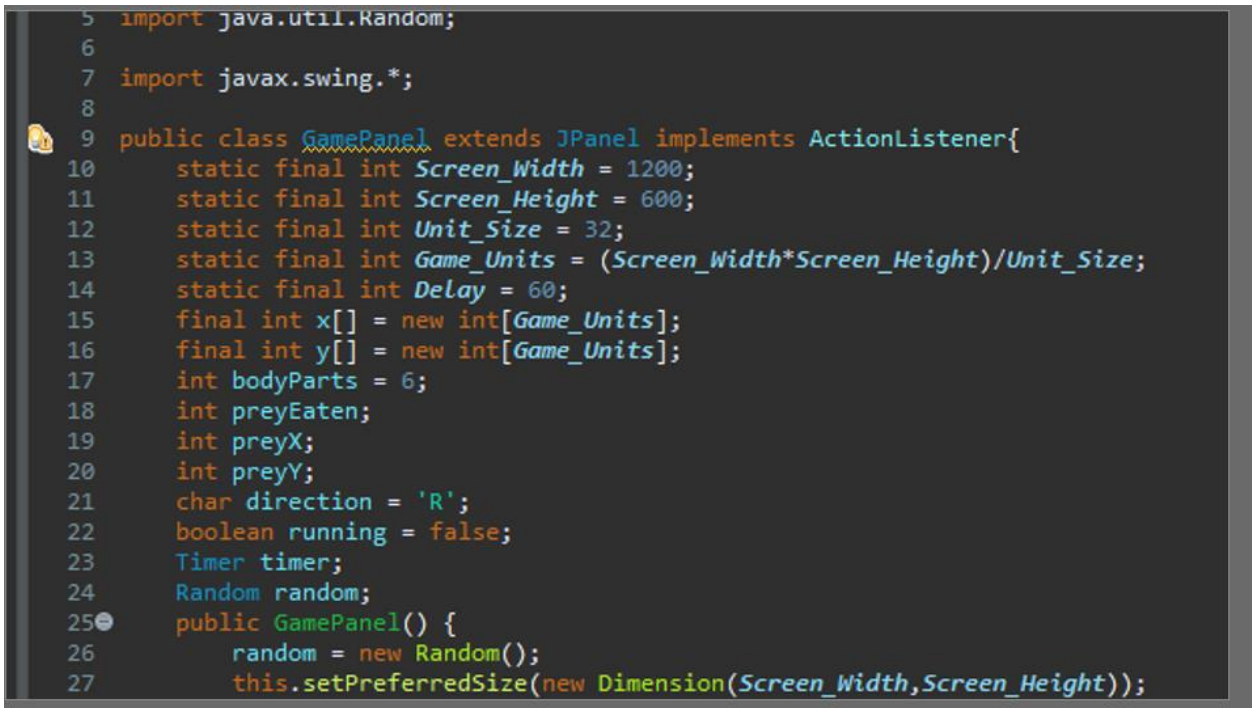
For our snake game we will be using the following:

event.ActionListener, event.ActionEvent, Event.KeyListener , event.KeyEvent

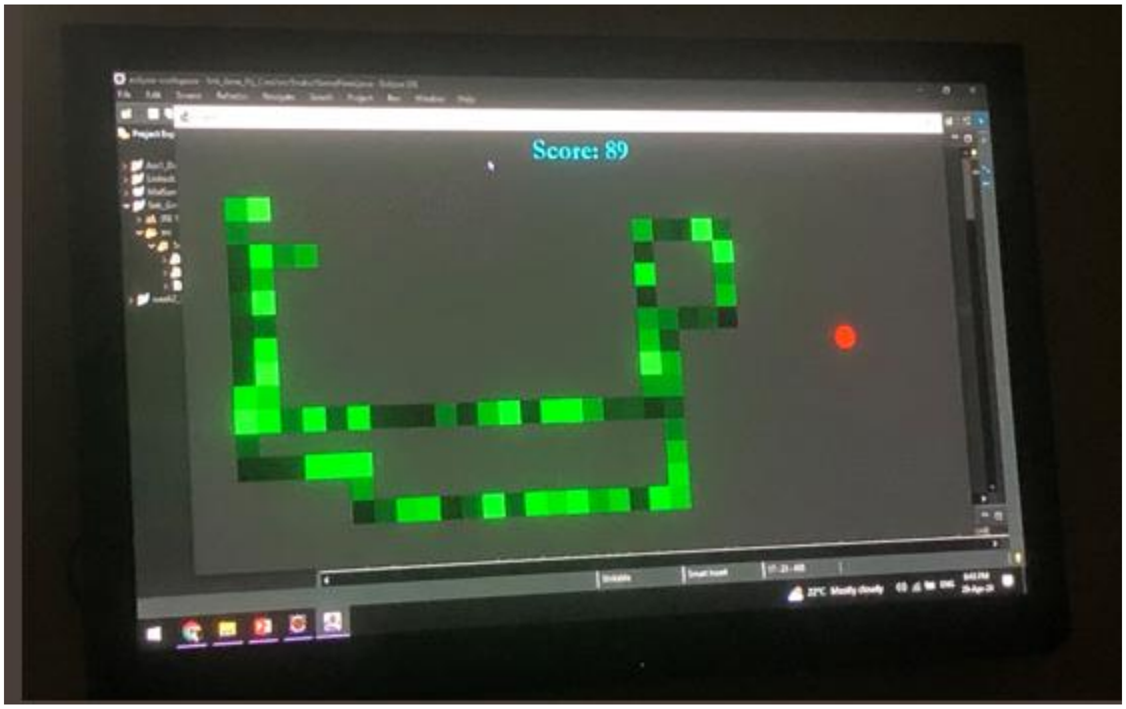
We will not be using the other classes since our input will only be 4 keyboard buttons: Up, down, left, right



```
package Snake;

import javax.swing.JFrame;

public class GameFrame extends JFrame {


public GameFrame() {

    this.add(new GamePanel());
    this.setTitle("Snake");
    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setResizable(false);
    this.pack();
    this.setVisible(true);
    this.setLocationRelativeTo(null);

}
```



```
import java.util.Random;

import javax.swing.*;

public class GamePanel extends JPanel implements ActionListener{
    static final int Screen_Width = 1200;
    static final int Screen_Height = 600;
    static final int Unit_Size = 32;
    static final int Game_Units = (Screen_Width*Screen_Height)/Unit_Size;
    static final int Delay = 60;
    final int x[] = new int[Game_Units];
    final int y[] = new int[Game_Units];
    int bodyParts = 6;
    int preyEaten;
    int preyX;
    int preyY;
    char direction = 'R';
    boolean running = false;
    Timer timer;
    Random random;
    public GamePanel() {
        random = new Random();
        this.setPreferredSize(new Dimension(Screen_Width,Screen_Height));
```



The rest of the code will be in separate files.

Part 2:

Mine craft development in java:

Research on Minecraft Java Edition's Custom Engine and its Mechanics

Introduction (wiki)

Minecraft Java Edition, developed by Mojang Studios, utilizes a custom-built engine that enables unique features and updates, such as redstone mechanics and diverse biomes, which have become iconic aspects of the game. This specialized engine offers a level of creativity, flexibility, and performance optimization that general-purpose engines like Unity struggle to achieve. Consequently, many games attempting to replicate Minecraft's success using engines like Unity have failed. This report delves into the specifics of Minecraft's engine, the logistics of game mechanics such as redstone, and how the custom engine facilitates major updates, while comparing it to the shortcomings of Unity-based clones.

Minecraft Java Edition's Custom Engine

Minecraft uses LWJGL (LightWeight java game librarY), a library to easily support OpenGL, openAL and keyboard/mouse-input in java. Minecraft is its own engine, and does not use a different engine.

Development and Features

Minecraft Java Edition's custom engine, primarily coded in Java, is known for its flexibility and performance optimization for the game's voxel-based world. Key features of this engine include:

- **Voxel Rendering:** Efficient rendering of the game's block-based world.

  The world is divided into chunks. From a chunk, the engine generates a triangle mesh, which is uploaded to a vertex buffer on the GPU. Occlusion queries are used to draw fewer chunks on-screen. Chunks are dynamically loaded and unloaded as you move around the world. (Vortex)

- **Modifiability:** Easy mod support due to Java's open-source nature.

  Easy mod support due to Java's open-source nature. This allows users to create and implement mods, expanding the game's capabilities and introducing new gameplay elements (Minecraft).

- **Cross-platform Compatibility:** Runs on multiple operating systems, including Windows, macOS, and Linux.

  (Cross-platform)

- **Networking Capabilities:** Robust support for multiplayer gaming with reliable networking protocols.

  This includes the ability to host and join servers, enabling a wide range of multiplayer experiences (Networking)

Architecture and Design

Minecraft Java Edition's engine is built around a unique architecture that prioritizes procedural generation, real-time world modification, and a block-based environment. The key components of this engine include:

1. **Procedural Generation:** The engine generates worlds procedurally, using a seed value to create vast, varied landscapes that include forests, caves, mountains, and oceans.
2. **Block System:** Every element in Minecraft is constructed from blocks, which can be manipulated individually, allowing players to modify the world dynamically.

3. **Optimization:** The engine employs advanced optimization techniques to handle large worlds and numerous entities efficiently, including chunk-based loading and rendering to manage resources and maintain performance.

Comparison with Unity-Based Clone: "Cube World"

"Cube World" developed by Picroma, attempted to recreate the Minecraft experience using the Unity engine. While Cube World had its merits, it ultimately failed to capture the essence and success of Minecraft. The reasons for its failure include:

1. **Engine Limitations:** Unity, as a general-purpose engine, struggled to handle the unique demands of a block-based, procedurally generated world as efficiently as Minecraft's custom engine.
2. **Performance Issues:** Cube World faced significant performance challenges, particularly in managing large, complex worlds and maintaining stable frame rates.
3. **Lack of Flexibility:** The Unity engine's more rigid structure made it difficult to implement the level of real-time world manipulation and extensive modding capabilities that Minecraft's custom engine supports.
4. **User Experience:** Lack of polish and frequent bugs diminished player experience.
5. **Mod Support:** Limited modifiability compared to Minecraft’s extensive modding community.

Game Mechanics and Major Updates

Core Mechanics

Minecraft's core mechanics revolve around exploration, resource gathering, crafting, and building. The game's open-ended nature encourages creativity and problem-solving. The procedural generation ensures that each world is unique, providing endless possibilities for exploration and construction.

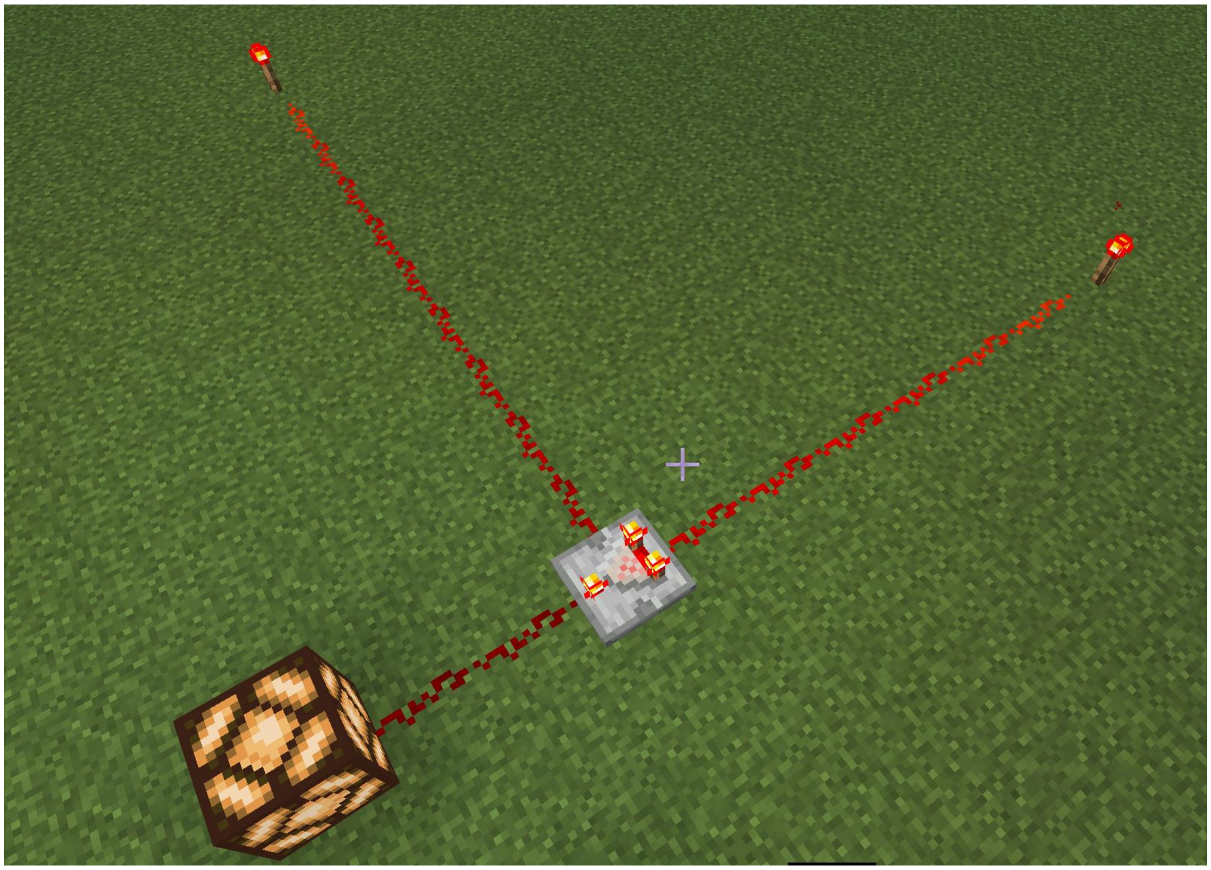
Redstone Mechanics

One of the most significant updates in Minecraft's history is the introduction of Redstone, which added complex circuitry and automation to the game. Redstone functions as an in-game material that can conduct power and create intricate mechanisms. The mechanics of redstone include:

1. **Redstone Dust:** Acts as wiring to connect components.
2. **Redstone Torches:** Serve as basic power sources and logic gates.
3. **Repeaters and Comparators:** Enhance and control redstone signals, enabling more complex circuits.

- **Power Source:** Redstone can transmit power to various devices such as doors, lamps, and pistons.
- **Circuitry:** Players can create simple to complex circuits using redstone dust, torches, repeaters, and comparators.
- **Automation:** Enables the creation of automated systems, such as farms and traps, enhancing the gameplay depth.

The custom engine's flexible architecture allowed seamless integration of these complex mechanics, something that engines like Unity would struggle with due to their general-purpose nature. These elements allow players to build anything from simple doors to complex computers and automated farms, adding a new dimension to gameplay and creativity.



Impact of the Custom Engine on Updates

Ocean Update

The custom engine's flexibility was pivotal in implementing the "Update Aquatic," which transformed the game's oceans. This update introduced new biomes, mobs (such as dolphins and turtles), and mechanics like swimming and underwater exploration. The engine's procedural generation was enhanced to create diverse and vibrant underwater environments, and new physics and rendering techniques were implemented to handle water dynamics and visibility.

Forest and Cave Biomes

The "Caves & Cliffs" update significantly altered Minecraft's subterranean and mountainous regions. The custom engine facilitated these changes by:

1. **Enhanced Procedural Generation:** Creating more varied and realistic cave systems and mountain structures.
2. **New Terrain Features:** Introducing elements like dripstone, lush caves, and deep dark biomes, each with unique gameplay mechanics and visual aesthetics.

3. **Improved World Generation Algorithms:** Ensuring smooth integration of new features with existing terrain, maintaining the game's coherence and performance.

Minecraft Java Edition's custom engine has been a cornerstone of its success, enabling unique features and continuous updates. In contrast, games like "Cube World" developed with Unity, failed to replicate Minecraft's success due to performance issues and lack of innovation. The custom engine not only supports complex mechanics like redstone but also allows for the seamless introduction of major updates, ensuring Minecraft's longevity and continued popularity.

Comparison: Minecraft Pocket Edition (MCPE) Windows 10 Edition vs. Java Edition

Minecraft is available in multiple versions, each tailored to different platforms and player experiences. Two prominent versions are the **Minecraft Pocket Edition (MCPE) Windows 10 Edition** and the **Java Edition**. Despite offering the same core gameplay, these versions have significant differences in terms of their underlying code, features, and performance.

Development and Features

Minecraft Pocket Edition (MCPE) Windows 10 Edition:

- **Programming Language**: Primarily coded in C++.
- **Cross-Platform Play**: Designed to support cross-play between different devices, including mobile, consoles, and Windows 10 PCs.
- **Performance Optimization**: Built to run smoothly on lower-end devices, focusing on efficient resource management and reduced load times.
- **Interface**: Touchscreen-friendly interface and controls adapted for mobile devices.

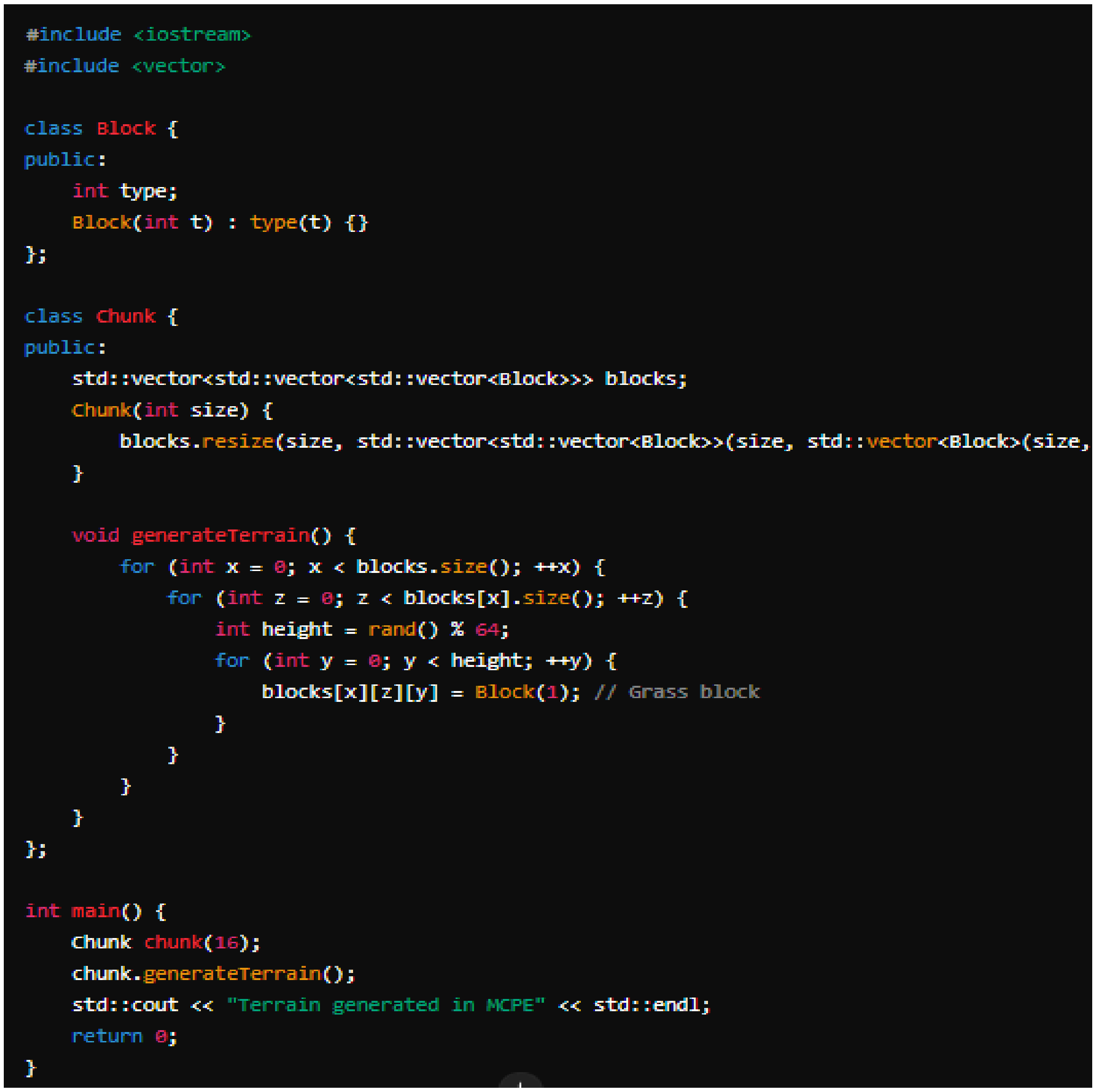
Minecraft Java Edition:

- **Programming Language**: Primarily coded in Java.
- **Mod Support**: Extensive modding community with a vast array of mods available due to Java's open-source nature.
- **Performance**: Optimized for PCs with higher specifications, allowing for more complex world generation and higher graphical fidelity.
- **Customization**: Greater flexibility in game customization, including server settings and game modifications.

Code Comparison

Code Example: MCPE Windows 10 Edition (C++)

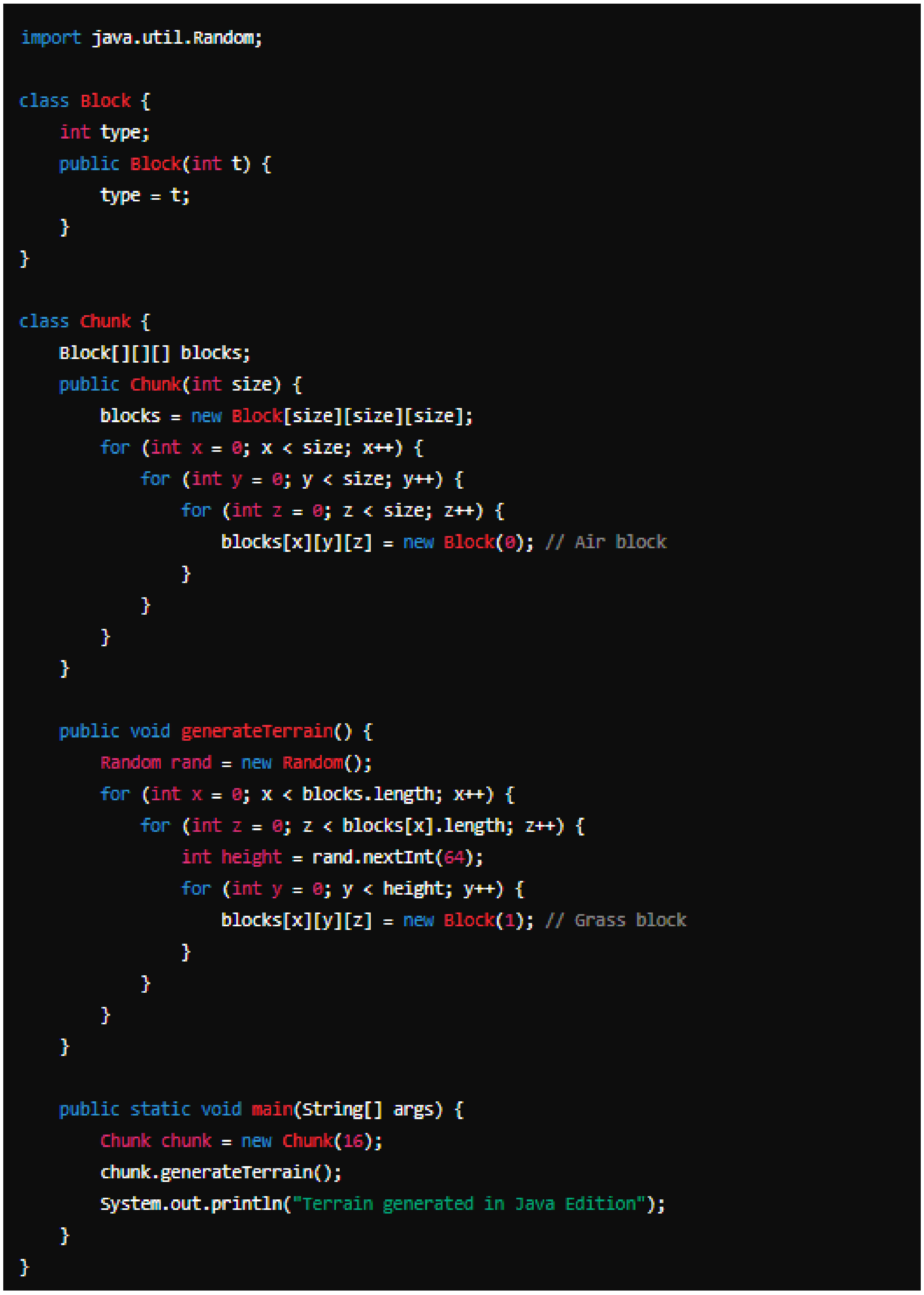
Here is a simplified example of code for generating terrain in MCPE Windows 10 Edition using C++:



```cpp
#include <iostream>
#include <vector>

class Block {
public:
    int type;
    Block(int t) : type(t) {}
};

class Chunk {
public:
    std::vector<std::vector<std::vector<Block>>> blocks;
    Chunk(int size) {
        blocks.resize(size, std::vector<std::vector<Block>>(size, std::vector<Block>(size,
    }

    void generateTerrain() {
        for (int x = 0; x < blocks.size(); ++x) {
            for (int z = 0; z < blocks[x].size(); ++z) {
                int height = rand() % 64;
                for (int y = 0; y < height; ++y) {
                    blocks[x][z][y] = Block(1); // Grass block
                }
            }
        }
    }
};

int main() {
    Chunk chunk(16);
    chunk.generateTerrain();
    std::cout << "Terrain generated in MCPE" << std::endl;
    return 0;
}
```

Example: Minecraft Java Edition (Java)

Her is a simplified example of code for generating terrain in Minecraft Java Edition:



```java
import java.util.Random;

class Block {
    int type;
    public Block(int t) {
        type = t;
    }
}

class Chunk {
    Block[][][] blocks;
    public Chunk(int size) {
        blocks = new Block[size][size][size];
        for (int x = 0; x < size; x++) {
            for (int y = 0; y < size; y++) {
                for (int z = 0; z < size; z++) {
                    blocks[x][y][z] = new Block(0); // Air block
                }
            }
        }
    }

    public void generateTerrain() {
        Random rand = new Random();
        for (int x = 0; x < blocks.length; x++) {
            for (int z = 0; z < blocks[x].length; z++) {
                int height = rand.nextInt(64);
                for (int y = 0; y < height; y++) {
                    blocks[x][y][z] = new Block(1); // Grass block
                }
            }
        }
    }

    public static void main(String[] args) {
        Chunk chunk = new Chunk(16);
        chunk.generateTerrain();
        System.out.println("Terrain generated in Java Edition");
    }
}
```

Performance and User Experience

Minecraft Pocket Edition (MCPE) Windows 10 Edition:

- **Performance**: Optimized for lower-end hardware, ensuring smooth gameplay on a wide range of devices.
- **User Experience**: Streamlined for touchscreen controls, with a user interface designed for ease of use on mobile devices.

Minecraft Java Edition:

- **Performance**: While capable of higher graphical settings and more complex modding, it can be more demanding on system resources.
- **User Experience**: Provides a more customizable and flexible gameplay experience, especially for players on PC who prefer using a keyboard and mouse.

Conclusion

While both the **Minecraft Pocket Edition (MCPE) Windows 10 Edition** and the **Java Edition** offer the quintessential Minecraft experience, they cater to different audiences and use cases. The MCPE Windows 10 Edition excels in accessibility and performance on a wide range of devices, making it ideal for casual gaming and cross-platform play. In contrast, the Java Edition stands out with its deep modding capabilities, flexibility, and performance on high-end PCs, appealing to dedicated gamers and modders.

Part 3:

Methodology of Rendering Architecture in Unity

Rendering in Unity encompasses a series of meticulously designed stages, each playing a crucial role in translating 3D scenes into 2D images. The architecture of Unity's rendering pipeline is crafted to be both flexible and efficient, utilizing the full potential of hardware and software capabilities. This detailed methodology outlines the key components and stages involved in Unity's rendering architecture, highlighting their professional intricacies. Scene Preparation Unity employs a scene graph to manage objects within a scene, structuring each object as a node in a hierarchical organization. This scene graph ensures that each node, representing an object, maintains its transform attributes (position, rotation, scale) relative to its parent, ultimately situating it in world space. This hierarchical structuring facilitates efficient transformations and object management. Culling is a critical optimization technique used to determine which objects lie within the camera's view frustum and are therefore eligible for rendering. Occlusion culling further refines this by eliminating objects obstructed by other geometry within the scene. Unity implements advanced techniques such as

Occlusion Queries and Precomputed Occlusion Data to perform these tasks, ensuring that only the visible and relevant objects consume rendering resources. Rendering Pipeline Stages Vertex Processing The rendering process initiates with vertex processing, where each vertex of a 3D model undergoes transformation via a vertex shader. This transformation shifts vertices from model space to screen space, effectively preparing them for further processing. For animated models, additional computations adjust vertex positions based on bone transformations, enabling complex character animations and dynamic deformations. Geometry Processing Following vertex processing, the geometry stage assembles vertices into geometric primitives—triangles, lines, or points—that the rasterizer will subsequently process. Unity supports tessellation shaders that can subdivide these primitives into finer geometry, allowing for more detailed and smoother surfaces. Rasterization Rasterization is the process of converting these geometric primitives into fragments, which are potential pixels on the screen. During this stage, depth testing is performed to determine the visibility of fragments, ensuring that only those fragments closer to the camera are rendered. Fragment Processing Fragment shaders are employed to compute the color and other attributes of each fragment based on material properties, lighting conditions, and textures. Unity supports a variety of lighting models, including the Phong and Physically Based Rendering (PBR) models, to achieve realistic shading effects. Texture sampling techniques such as filtering and mipmapping are used to enhance visual quality and reduce artifacts. Post-processing effects play a significant role in the final appearance of the rendered image. Unity's Post-Processing Stack includes a variety of effects, such as bloom, motion blur, color grading, and anti-aliasing, which are applied to refine and polish the final output. Screen-space effects like Screen-Space Ambient Occlusion (SSAO) and Screen-Space Reflections (SSR) add an additional layer of realism by simulating complex lighting interactions. Lighting and Shading Unity's lighting system supports various light types to cater to different scenarios and artistic needs. Directional lights simulate distant light sources like the sun, providing consistent lighting across large areas. Point lights emit light in all directions from a single point, ideal for localized lighting effects. Spotlights emit light in a cone shape, suitable for focused illumination in scenes. Shadow mapping is an integral part of lighting in Unity. The engine generates shadow maps for lights that cast shadows, using techniques such as Cascaded Shadow Maps for directional lights to handle large-scale outdoor environments efficiently. Screen-space shadows are applied during post-processing to enhance shadow detail and realism, ensuring that shadows blend seamlessly with the surrounding environment. Render Targets and Buffers The final rendered image is stored in the framebuffer, which the GPU then displays on the screen. Intermediate rendering results can be stored in render textures, enabling advanced effects like reflections and deferred shading. Deferred shading is a technique where surface properties are first written to multiple render targets, known as G-buffers, during the geometry pass. Lighting calculations are then performed in screen space using the information stored in the G-buffers. This approach allows for efficient handling of complex lighting scenarios and multiple light sources. Optimization Techniques Optimization is crucial for maintaining performance, especially in complex scenes. Unity's Level of Detail (LOD) systems reduce the complexity of distant objects, conserving rendering resources. Beyond basic frustum and occlusion culling, Unity supports impostors and other advanced techniques to further optimize performance. Batch rendering is another optimization strategy that combines multiple objects into a single draw call, reducing CPU overhead and improving rendering efficiency. Instancing allows for the efficient rendering of many instances of the same geometry with different transformations and materials, significantly enhancing performance in scenes with repetitive objects. Platform-Specific Considerations Unity supports multiple graphics APIs, including DirectX, OpenGL, Vulkan, and Metal, to leverage platform-specific optimizations. On Apple devices, Unity uses Metal for efficient rendering, ensuring that the engine takes full advantage of the hardware capabilities. For mobile optimization, Unity employs shader variants to optimize performance across different devices. Adaptive quality dynamically adjusts rendering quality based on performance metrics, ensuring a balance between visual fidelity and smooth performance on various hardware configurations. Custom Shaders and Scriptable Render Pipeline (SRP) Unity's Shader Graph provides a visual interface for creating shaders, allowing developers to design complex shading effects without writing code. For those requiring more control, custom shaders can be written in HLSL, offering precise manipulation of rendering processes. The Scriptable Render Pipeline (SRP) framework includes the Universal Render Pipeline (URP) and the High Definition Render Pipeline (HDRP). URP is designed for flexibility and optimization across a wide range of platforms, while HDRP targets high-end hardware, offering advanced rendering techniques and photorealistic visuals. Developers can create custom render pipelines tailored to specific requirements, providing unmatched flexibility in rendering workflows. Conclusion Unity's rendering architecture is a sophisticated and highly optimized system designed to handle a diverse array of graphics scenarios. By leveraging a combination of culling techniques, shader processing, lighting models, and post-processing effects, Unity efficiently renders scenes across various platforms. The flexibility offered by custom shaders and the Scriptable Render Pipeline ensures that developers have the tools necessary to achieve their desired visual outcomes, making Unity a powerful engine for both simple and complex rendering tasks

Part4

Stardew valley case study(Building your own engine on visual studio)

Understanding such structures :

Creating a successful game like Stardew Valley using Visual Studio involves a series of methodical steps, including planning, development, testing, and iteration. Here's a summarized methodology:

1. **Planning and Conceptualization**:
   - Define the game concept, genre, and core mechanics. Stardew Valley was inspired by classic farming simulation games.
   - Create a design document outlining gameplay, story, characters, and art style.
2. **Setting Up the Development Environment**:

- Install Visual Studio, a versatile IDE that supports various programming languages.
- Choose the right programming language and game engine. Stardew Valley was developed using C# and the XNA framework, which is integrated with Visual Studio.

3. **Prototyping**:
   - Develop a basic prototype to test core mechanics and gameplay ideas.
   - Focus on key features like farming, crafting, and NPC interactions, ensuring they are fun and engaging.
4. **Core Development**:
   - Implement game features iteratively, starting with the most critical aspects.
   - Use Visual Studio's debugging tools to troubleshoot and optimize code.
   - Maintain a clean and organized codebase to facilitate future updates and modifications.
5. **Graphics and Art Development**:
   - Create or commission game assets, including sprites, tilesets, and animations.
   - Integrate these assets into the game, ensuring they align with the overall aesthetic.
6. **Testing and Quality Assurance**:
   - Conduct thorough testing, both internally and through beta testing with a wider audience.
   - Use Visual Studio's testing tools to identify and fix bugs, improve performance, and refine gameplay.
7. **Iteration and Feedback Integration**:
   - Gather feedback from testers and players to make informed adjustments and improvements.
   - Continuously update and polish the game based on feedback and testing results.
8. **Release and Post-Launch Support**:
   - Prepare for the game's launch by marketing, creating promotional materials, and engaging with the community.
   - Release the game on various platforms, utilizing Visual Studio's support for different operating systems.
   - Provide ongoing support and updates to address any issues and add new content.

By following these steps, the developer of Stardew Valley was able to create a highly successful game, leveraging the robust features of Visual Studio to streamline the development process. Refrence: Eric Barone(wiki)

Part5: (Creating your own engine)

To explain the different classes or packages found in a common visual studio developed game in order to have your own foundations for optimized methods and game abilities we have to look at the common structure:

In a typical game developed in Visual Studio, especially one using C# and a framework like XNA or MonoGame, the project's structure is organized into classes and packages (or namespaces). Here's an outline of common classes and packages you might find:

Common Packages/Namespaces

1. **Core**
   - **Game1.cs**: The main class that inherits from the Game class. It manages the game's lifecycle, including initialization, updating, and rendering.
   - **Program.cs**: The entry point of the application, containing the Main method that starts the game.
2. **Graphics**
   - **Sprite.cs**: Manages the loading and rendering of 2D sprites.
   - **Animation.cs**: Handles sprite animations, including frame updates and timing.
3. **Input**
   - **InputManager.cs**: Centralized management of user input (keyboard, mouse, gamepad).
   - **KeyboardInput.cs**: Specific handling of keyboard input.
   - **MouseInput.cs**: Specific handling of mouse input.
4. **Audio**
   - **SoundManager.cs**: Manages sound effects and background music.
   - **SoundEffect.cs**: Handles individual sound effects.
   - **Music.cs**: Handles background music tracks.
5. **Entities**
   - **Player.cs**: Defines player-specific properties and behaviors.
   - **NPC.cs**: Defines non-player character properties and behaviors.
   - **Enemy.cs**: Defines enemy properties and behaviors.
   - **Item.cs**: Manages in-game items, their properties, and interactions.
6. **World**
   - **TileMap.cs**: Manages the game's tile-based map.
   - **Tile.cs**: Represents individual tiles on the map.
   - **Environment.cs**: Manages environmental elements like weather, time of day, etc.
7. **UI (User Interface)**
   - **UIManager.cs**: Central management of all UI elements.
   - **Button.cs**: Represents interactive button elements.
   - **Label.cs**: Represents text labels.
   - **Menu.cs**: Manages in-game menus and their elements.
8. **Systems**

- **Physics.cs**: Manages physics calculations, including collisions and movement.
- **Collision.cs**: Handles collision detection and response.

9. **Utilities**
    - **Logger.cs**: Manages logging for debugging and error tracking.
    - **Settings.cs**: Handles game settings and configurations.
    - **HelperFunctions.cs**: Miscellaneous utility functions used throughout the game.
10. **Content**
    - **ContentLoader.cs**: Manages the loading of game content, including textures, sounds, and other assets.

Example Project Structure

css

Copy code

```
MyGameProject
│
├── Core
│   ├── Game1.cs
│   ├── Program.cs
│
├── Graphics
│   ├── Sprite.cs
│   ├── Animation.cs
│
├── Input
│   ├── InputManager.cs
│   ├── KeyboardInput.cs
│   ├── MouseInput.cs
│
├── Audio
│   ├── SoundManager.cs
│   ├── SoundEffect.cs
│   ├── Music.cs
│
├── Entities
│   ├── Player.cs
│   ├── NPC.cs
│   ├── Enemy.cs
│   ├── Item.cs
│
├── World
│   ├── TileMap.cs
│   ├── Tile.cs
│   ├── Environment.cs
│   │
```

```
│
├── UI
│   ├── UIManager.cs
│   ├── Button.cs
│   ├── Label.cs
│   ├── Menu.cs
│
├── Systems
│   ├── Physics.cs
│   ├── Collision.cs
│
├── Utilities
│   ├── Logger.cs
│   ├── Settings.cs
│   ├── HelperFunctions.cs
│
├── Content
│   ├── ContentLoader.cs
│
├── Content (Folder)
│   ├── Textures
│   ├── Sounds
│   ├── Music
```

Detailed Class Descriptions

- **Game1.cs**: Inherits from the Game class and serves as the main game loop, managing the initialization, updating, and drawing of the game.
- **Program.cs**: Contains the Main method, which is the entry point of the application.
- **Sprite.cs**: Manages 2D image rendering, including position, rotation, and scaling.
- **Animation.cs**: Handles frame-based animations for sprites.
- **InputManager.cs**: Manages and processes input from various devices.
- **SoundManager.cs**: Centralizes sound effect and music playback.
- **Player.cs**: Represents the player character, including movement, actions, and interactions.
- **TileMap.cs**: Manages the game's map, including loading, rendering, and interaction with tiles.
- **UIManager.cs**: Manages all user interface elements, ensuring they are correctly displayed and interactable.
- **Physics.cs**: Handles physical interactions, such as collisions and movements.
- **Logger.cs**: Provides logging functionality to help with debugging and tracking errors.
- **ContentLoader.cs**: Manages loading and organizing game assets, such as textures, sounds, and music.

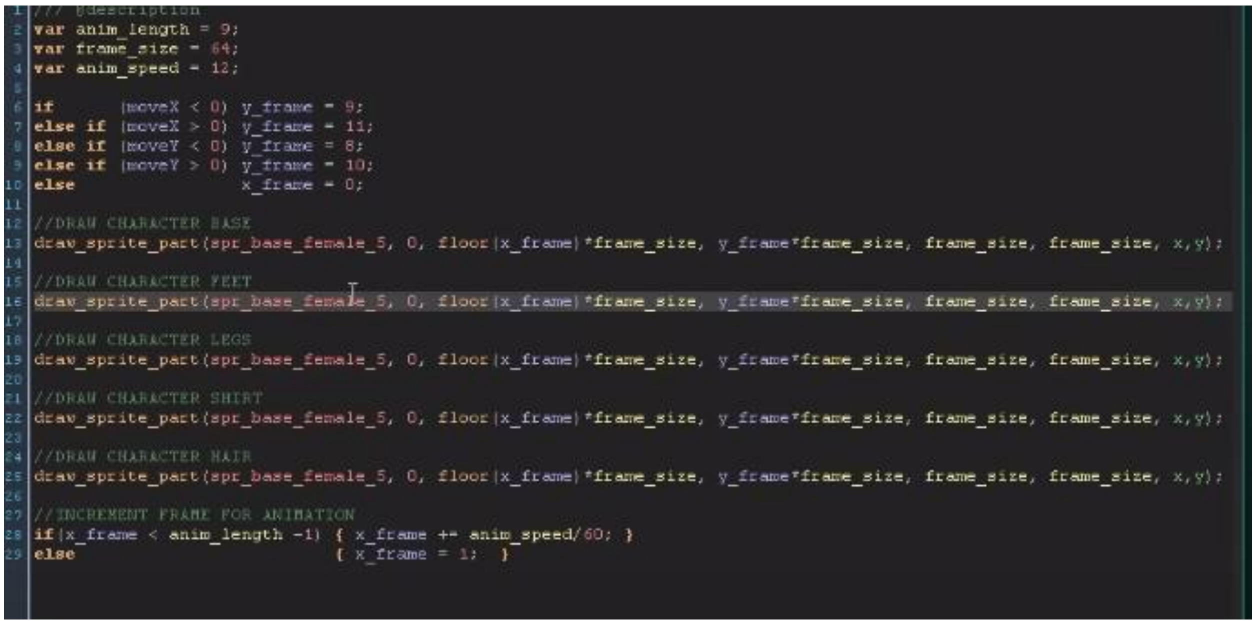
By organizing the game into these classes and packages, developers can manage the complexity of game development, ensuring a modular and maintainable codebase.

Conclusion

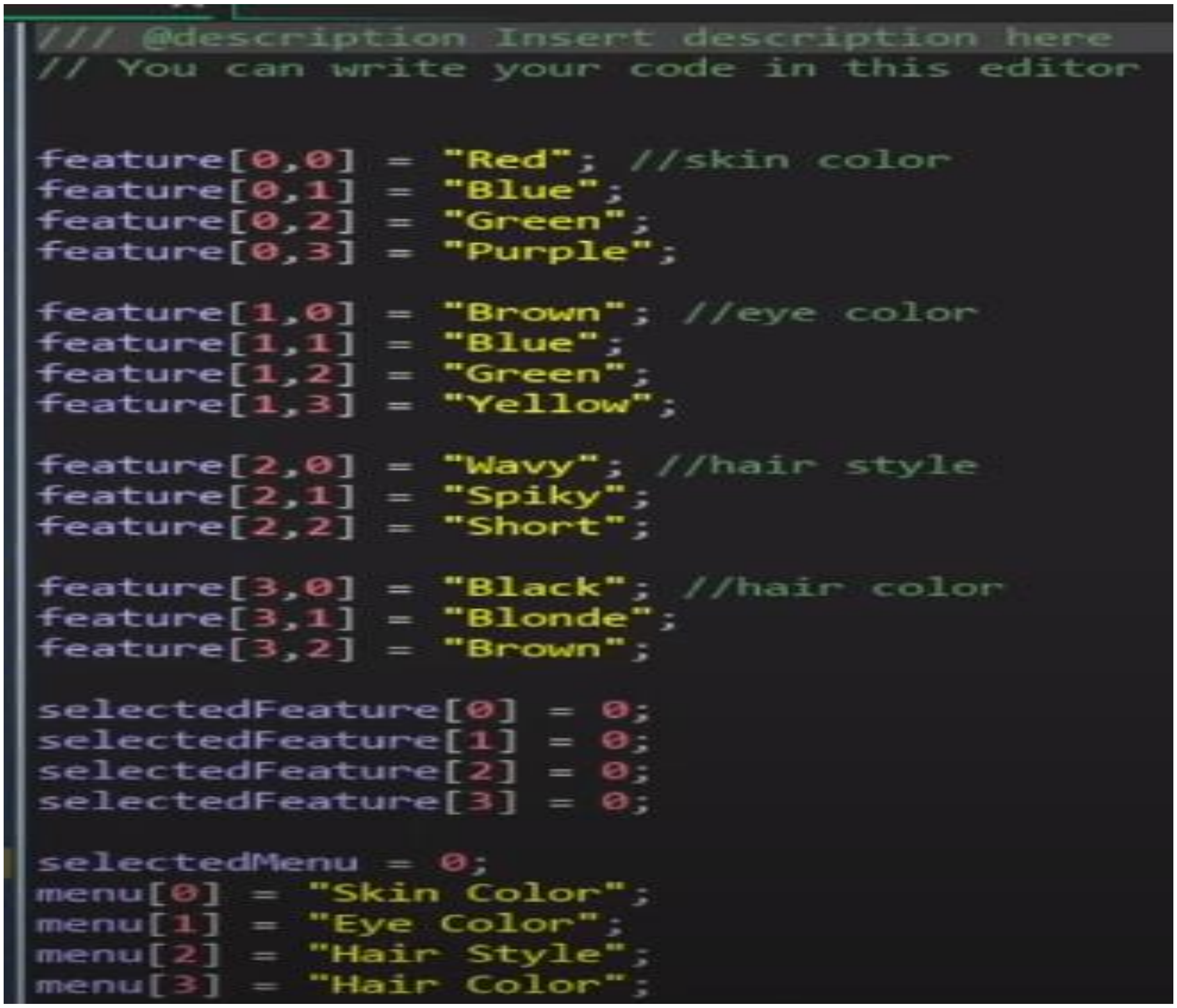
For beginners, **Unity** is often considered the best choice due to its balance of ease of use, extensive learning resources, and flexibility in creating both 2D and 3D games. However, if you are more interested in high-quality graphics and visual scripting, **Unreal Engine** is an excellent option. For those who prefer a more lightweight and beginner-friendly approach, **Godot** offers a great entry point, especially with its easy-to-learn scripting language. Lastly, **GameMaker Studio** is perfect for those focused on 2D game development and looking for a simple drag-and-drop interface.

Ultimately, the best game engine for you will depend on your specific needs, goals, and preferences. It's a good idea to try out a few different engines to see which one you find the most intuitive and enjoyable to use.

Game maker example from my own game using the GML language(Method used for pallet swapping and rendering sprite sheet based on 4 different directions:



```
/// @description
var anim_length = 9;
var frame_size = 64;
var anim_speed = 12;

if      (moveX < 0) y_frame = 9;
else if (moveX > 0) y_frame = 11;
else if (moveY < 0) y_frame = 8;
else if (moveY > 0) y_frame = 10;
else                x_frame = 0;

//DRAW CHARACTER BASE
draw_sprite_part(spr_base_female_5, 0, floor(x_frame)*frame_size, y_frame*frame_size, frame_size, frame_size, x,y);

//DRAW CHARACTER FEET
draw_sprite_part(spr_base_female_5, 0, floor(x_frame)*frame_size, y_frame*frame_size, frame_size, frame_size, x,y);

//DRAW CHARACTER LEGS
draw_sprite_part(spr_base_female_5, 0, floor(x_frame)*frame_size, y_frame*frame_size, frame_size, frame_size, x,y);

//DRAW CHARACTER SHIRT
draw_sprite_part(spr_base_female_5, 0, floor(x_frame)*frame_size, y_frame*frame_size, frame_size, frame_size, x,y);

//DRAW CHARACTER HAIR
draw_sprite_part(spr_base_female_5, 0, floor(x_frame)*frame_size, y_frame*frame_size, frame_size, frame_size, x,y);

//INCREMENT FRAME FOR ANIMATION
if(x_frame < anim_length -1) { x_frame += anim_speed/60; }
else                         { x_frame = 1;  }
```



```
/// @description Insert description here
// You can write your code in this editor


feature[0,0] = "Red"; //skin color
feature[0,1] = "Blue";
feature[0,2] = "Green";
feature[0,3] = "Purple";

feature[1,0] = "Brown"; //eye color
feature[1,1] = "Blue";
feature[1,2] = "Green";
feature[1,3] = "Yellow";

feature[2,0] = "Wavy"; //hair style
feature[2,1] = "Spiky";
feature[2,2] = "Short";

feature[3,0] = "Black"; //hair color
feature[3,1] = "Blonde";
feature[3,2] = "Brown";

selectedFeature[0] = 0;
selectedFeature[1] = 0;
selectedFeature[2] = 0;
selectedFeature[3] = 0;

selectedMenu = 0;
menu[0] = "Skin Color";
menu[1] = "Eye Color";
menu[2] = "Hair Style";
menu[3] = "Hair Color";
```

These are simple examples on how game engines can have certain features that will make it easier to implement inventory systems and sprite exchanges without having the need to develop a database for the objects simply cross referencing the sprite sheets.

This is the basic knowledge needed to start the journey, the tools are different but the conceptual methodology remains the same. This is why game engines are recommended for beginners, since they give the knowledge of how to architect your game concepts into a feasible reality.